个性化你的阅读

NO.20

编程狂人

Programming Madman

精华版

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法,从海量文章资讯中挖掘出高质量的内容,并通过分析用户的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容, 满足你日常的专业阅读需要。我们针对IT人还做了个活动频道,它聚合了 IT 圈 最新最全的线上线下活动, 使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊,目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选整理出来。 每期的周刊一般会周二的某个时间点发布。

本期为精简版周刊完整版链接:
http://www.tuicool.com/mags/534aab89d91b141dac0c55ce

版权说明

欢迎大家扫描下载推酷客户端

目录

TIOBE 2014年4月编程语言排行榜：PERL跌至历史最低点

TIOBE2014年4月份编程语言排行榜出炉，尽管前三甲排名无变化为：C、Java、Objective-C，但是C和Java的份额均有所下降，但是幅度不明显；而Objective-C则上涨了3.28%。

这个月Perl语言下降幅度较大。此前，我们曾说过如果Perl再不发力，以这种趋势发展下去兴许会跌出Top 10。果然这个月跌出TOP10，排名13位，跌入历史最低点。Perl加油啊！

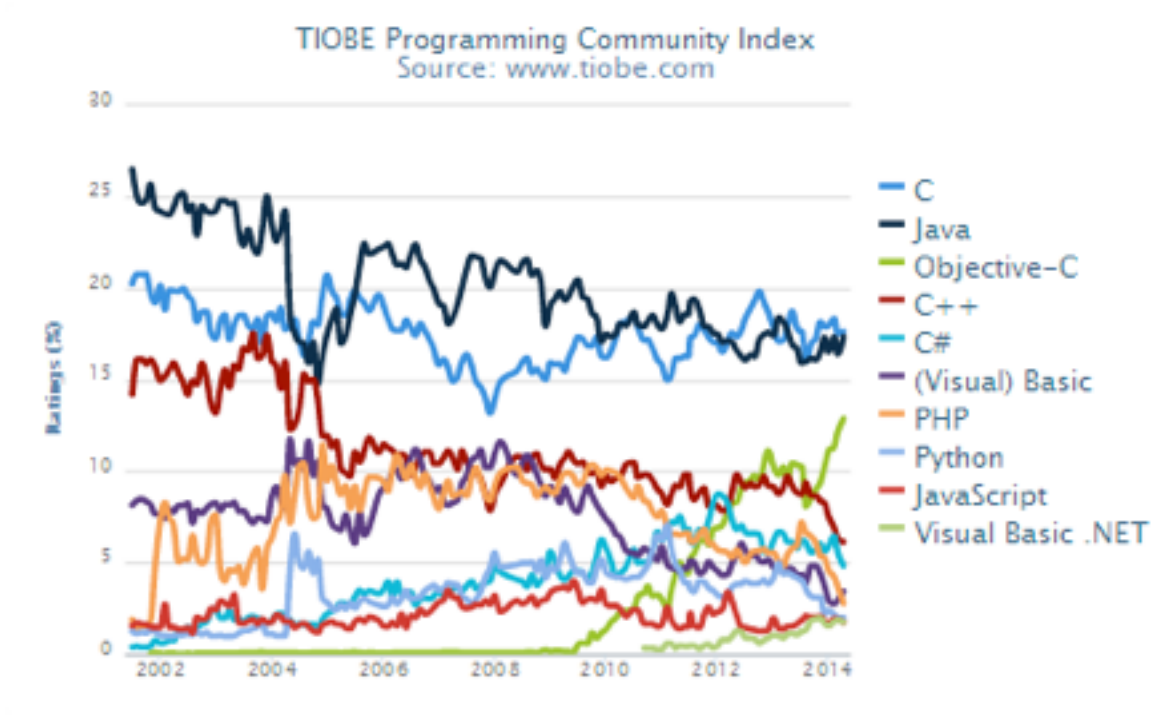
设计者Larry Wall为了让在UNIX上进行报表处理的工作变得更方便，决定开发一个通用的脚本语言，于是Perl孕育而生。Perl是一种高级、通用、直译式、动态的程序语言，很多开发者喜欢它的理由是因其具备强力、稳定、成熟、可移植性等特点。

值得一提的是，Larry Wall在OSTC大会分享其编程人生（PPT）。感兴趣的朋友可以移步去看下。

2014年4月编程语言排行榜TOP 20榜单：

Apr 2014	Apr 2013	Change	Programming Language	Ratings	Change
1	1		C	17.631%	-0.23%
2	2		Java	17.348%	-0.33%
3	4	^	Objective-C	12.875%	+3.28%
4	3	v	C++	6.137%	-3.58%
5	5		C#	4.820%	-1.33%
6	7	^	(Visual) Basic	3.441%	-1.26%
7	6	v	PHP	2.773%	-2.65%
8	8		Python	1.993%	-2.45%
9	11	^	JavaScript	1.750%	+0.24%
10	12	^	Visual Basic .NET	1.748%	+0.65%
11	10	v	Ruby	1.745%	-0.23%
12	17	^^	Transact-SQL	1.170%	+0.45%
13	9	vv	Perl	1.027%	-1.31%
14	52	^^	F#	0.966%	+0.83%
15	19	^^	Assembly	0.853%	+0.14%
16	13	v	Lisp	0.797%	-0.11%
17	18	^	PL/SQL	0.782%	+0.07%
18	24	^^	MATLAB	0.760%	+0.24%
19	15	vv	Delphi/Object Pascal	0.746%	-0.09%
20	35	^^	D	0.708%	+0.39%

前10名编程语言长期走势图：



以下是21-50编程语言排名：

Position	Programming Language	Ratings
21	Logo	0.693
22	SAS	0.688
23	ML	0.643
24	Pascal	0.598
25	PostScript	0.578
26	OpenEdge ABL	0.539
27	ActionScript	0.438
28	PL/I	0.434
29	COBOL	0.429
30	Ladder Logic	0.380
31	ABAP	0.367
32	cT	0.360
33	C shell	0.352
34	Fortran	0.347
35	Lua	0.332

36	Go	0.324
37	Ada	0.287
38	Scratch	0.283
39	Emacs Lisp	0.269
40	R	0.265
41	Z shell	0.253
42	Groovy	0.247
43	S-PLUS	0.241
44	Io	0.229
45	Tcl	0.226
46	NXT-G	0.225
47	JScript.NET	0.220
48	Scala	0.219
49	Prolog	0.209
50	OCaml	0.197

后50名编程语言如下：

- (Visual) FoxPro, 4th Dimension/4D, Arc, ATLAS, Automator, Avenue, Awk, Bash, BlitzMax, Bourne shell, cg, CL (OS/400), Clean, Common Lisp, Erlang, Factor, Felix, Forth, Haskell, Icon, Inform, Informix-4GL, J, JavaFX Script, Korn shell, LabVIEW, M4, Magic, Max/MSP, Mercury, Modula-2, Moto, NATURAL, OpenCL, PILOT, Programming Without Coding Technology, Pure Data, Q, Revolution, RPG (OS/400), S, Scheme, Slate, Smalltalk, SPARK, Standard ML, TOM, VBScript, VHDL, X10

必须声明，这个榜单本身采集的是英文世界的数据，虽然在反映趋势上有一些参考意义，但与中国的实际情况不完全符合，而且，这张采样本身也有相当大的局限性。

【说明】

TIOBE编程语言社区排行榜是编程语言流行趋势的一个指标，每月更新，这份排行榜排名基于互联网上有经验的程序员、课程和第三方厂商的数量。排名 使用著名的搜索引擎（诸如Google、MSN、Yahoo!、Wikipedia、YouTube以及Baidu等）进行计算。请注意这个排行榜只是反 映某个编程语言的热门程度，并不能说明一门编程语言好不好，或者一门语言所编写的代码数量多少。

这个排行榜可以用来考查你的编程技能是否与时俱进，也可以在开发新系统时作为一个语言选择依据。排行榜的详细定义可以参考这里（英文）。

作者：夏梦竹

本文转载自：http://www.csdn.net/article/2014-04-10/2819229-TIOBE-Programming

为什么FACEBOOK发明新语言“HACK”

(注：Hack是一种PHP的派生语言）

为了替换掉那些有年头的老代码，Facebook创建了一个新的语言。这篇文章将会告诉背后的故事。

By Steven Melendez

这 个故事来自Facebook工程师Julien Verlaguet和Ed Smith的一次访谈。2004年2月，扎克伯格（Mark Zuckerberg）的哈佛同学们第一次登陆Facebook，服务器里运行着PHP。那个时候PHP击败了Perl成为最炙手可热的 Web开发语言。

使用类似Ruby的Rails或者Python的Django这些当下最流行的开发框架都不是那个时候的选择。Rails 第一次发布是在几个月之后（注：2004年7月），而Django在一年后才被发布（注：2005年）。十年后，PHP因为笨拙的库，不一致的命名规则， 内置定义的函数，在多语言程序中语法和语义与相关语言巨大差异使人混淆，过去的设计原则极容易导致安全性问题等而被指责。

“每一个PHP程序员日复一日地处理着难以捉摸或者棘手的任务”，Facebook的工程师Julien Verlaguet和Alok Menghrajani在最近的公司博客上写道。

但是，PHP并没离开Facebook，并且其他的大公司和工程的数百万行代码都是用这个语言写的。程序员们仍然得益于PHP的快速开发和部署，且努力去除那些不好的特性。

减轻PHP程序员的痛苦不是说抛弃这个语言和多年的开发成果。Facebook开发了Hack，这是一种新的，派生于PHP的语言。它将与已有的代码和谐相处，增强了安全特性源于函数式编程语言和学术研究。

“这 是一种很特别的设计，能与PHP进行无缝的交互”，Verlaguet说。技术引领着Hack项目的发展，它的背后是一份混合编程语言和行业经验的正式学 术研究。Facebook已经在内部使用和发展Hack大概有两年时间了。最近，已经将项目开源，并且在4月9日安排公开的“开发者日”。

“我们这样做的目的是希望能更好地倾听来自社区的反馈， 同时开源社区也会让Hack面对Facebook外的开发者有更好的体验”，Verlaguet说。

也许Hack的主要创新是引入了自动类型判断，概念类似于深奥难懂的Haskell和 ML语言但是比他们少得多的命令行，同时更接近主流的编程语言。

传统的PHP是动态类型，这意味着在代码中的基本的本质的变量类型是一个数还是一个字符串或者其他类型是不确切的，除非程序实际运行着。程序员们享受这种灵活，却为错误开辟了空间，它不像Java或者C那样的静态类型语言，代码被写出来的时候就明确告诉你变量的类型。

Hack 走了一条中间路线：它可以基于变量怎样被使用的使用逻辑让开发者指定类型，如果代码的逻辑冲突，就会给出一个错误（error）。这个概念本身不是新的， 但是它以前都是被用在编译型语言上的，开发者需要等待他们的源代码被转换成机器码，不能像PHP程序员希望的那样点完保存立即执行，Verlaguet 说。

“解决方案在于建立了一个类型检查守护进程”，他说。关于这个后台程序运行在开发者的电脑里。它代替等待开发者去显示调用一个编译器，当源代码文件被改变的时候，类型检查进程要求操作系统去通知编译器。这类似于同步文件需要更新时，Dropbox就得到一个信号。

被改变检查的有效的方法是通过类型检查器被反复检测，直到其确认与其他的代码是一致的。只要类型检查器足够快，程序员基本就不需要等待，类似于在Git版本管理系统上转换到新分支那样，Verlaguet说。

Hack 还有其他一些特性，增强的集合类型如vector和set来增强PHP的数组，匿名函数被使用在函数编程上。新的语言让Facebook逐步提升已有的 PHP代码，使得长期投资在PHP上的价值继续发挥作用， Ed Smith说，Facebook的HHVM运行引擎将会同时提供给Hack和PHP。

“Hack 让我们能在同一时间和同一文件上动态转换我们的代码”，Smith说，“换做其他的一种语言将会有很大的困难”你认为是吗?

其他的公司和工程转换到时髦的Hack还为时尚早，项目刚刚开源，Verlaguet说。不过，从对他采访的记录来看，整个项目目前是处于积极的态势中。

英文原文：fastcolabs

文章转载自：开源中国社区 [http://www.oschina.net]
本文地址：http://www.oschina.net/news/50565/why-facebook-invented-a-new-php-derived-language-called-hack

如何高效地管理网站静态资源

作者/walter
前端工程师
出家如初,成佛有余

背景

随着互联网开发和迭代速度越来越快，网站也变得越来越庞大，存在大量静态资源，我们原有管理静态资源的方式变得越来越不适用，就如同封面图一样，静态资源之间的关系错综复杂，给工程师带来了很多麻烦：

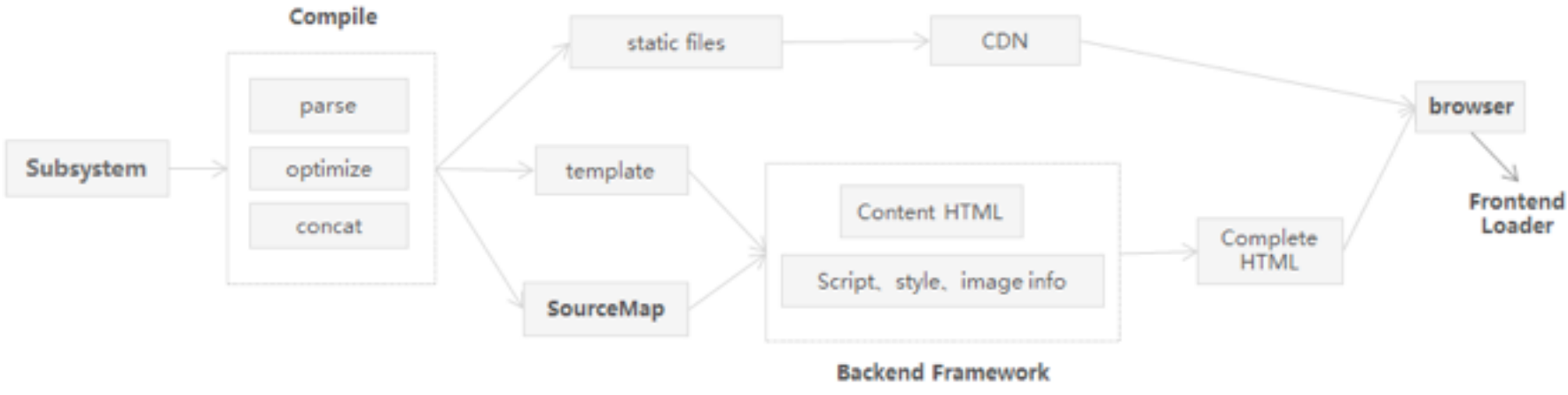
- 人工管理依赖的噩梦，工程师需要频繁管理和维护每个页面需要的 JS & CSS 文件，包括静态资源之间的依赖关系以及加载顺序等。
- 性能优化成本高且不可持续性，为了提高网站性能，工程师总是在忙于优化页面静态资源的加载，包括动态加载静态资源、按需加载静态资源和修改静态资源合并策略等，但是过了一段时间性能又降下来了，又需要周而复始的重复。
- 静态资源差异化的挑战，PC和无线的适配，不同的网络和终端需要适配相应的静态资源；当网站需要支持国际化的时候，需要对不同的国家进行差异化处理，返回不同的静态资源，这些需求对原有的静态资源管理方式提出巨大挑战。
- 缺少快速迭代和试验新功能的有效支持，从开发到上线流程繁琐，导致项目迭代周期长

每天工程师都会提交大量的 new feature/bug fixes，每次项目发布和迭代都面临着以上的问题，是否可以有一套系统帮助我们管理/调度静态资源来减少人工管理静态资源成本和风险，来达到更快、更可 靠、低成本的自动化项目交付。在实际项目开发中，我们进行了大量探索和试验，实现了一套 “静态资源管理系统”，对静态资源进行全流程的管理和调度：

- 帮助工程师管理静态资源间的依赖以及资源的加载
- 管理静态资源版本更新与缓存，自动处理CDN
- 自动生成最优的静态资源合并策略，实现网站自适应优化
- 实现静态资源的分级发布，快速迭代，轻松回滚
- 根据国际化和终端的差异，送达不同的资源给不同的用户

下面本文将会介绍我们是如何通过静态资源系统来高效管理静态资源的。

架构



静态资源管理系统主要包含Compile、Sourcemap、Backend-Framework、Frontend-Loader几个核心模块：

- Compile，对静态资源进行编译处理，包括对静态资源进行预处理，url 处理(添加md5戳、添加CDN前缀)，优化(压缩、合并)，生成 Sourcemap 等
- Sourcemap，在 compile 阶段系统会扫描静态资源，建立一张静态资源关系表，记录每个静态资源的部署路径以及依赖关系等信息
- Backend-Framework，后端运行时根据组件使用情况来调度静态资源，为前端返回页面渲染需要的资源。
- Frontend-Loader，前端运行时根据用户的交互行为动态请求静态资源。

静态资源管理系统通过自动化工具对静态资源进行预处理并产出 Sourcemap，Source-Map 中记录着静态资源的调度信息，这样框架在运行时会根据 SourceMap 中提供的调度信息自动为用户进行静态资源调度，不仅可以做到送达不同资源给不同用户，还可以自适应优化静态资源合并和加载。

自动管理静态资源依赖

静态资源管理系统为工程师提供了声明依赖关系的语法和规则，在 compile 阶段系统会扫描静态资源，建立一张静态资源关系表，记录每个静态资源的部署路径以及依赖关系等信息。

在html中声明依赖

在项目的 index.html 里使用注释声明依赖关系：

```
<!--
    @require demo.js
    @require "demo.css"
-->
```

在 SourceMap 中则可看到：

```
{
    "res" : {
        "demo.css" : {
            "uri" : "/static/css/demo_7defa41.css",
            "type" : "css"
        },
        "demo.js" : {
            "uri" : "/static/js/demo_33c5143.js",
            "type" : "js",
            "deps" : [ "demo.css" ]
        },
        "index.html" : {
            "uri" : "/index.html",
            "type" : "html",
            "deps" : [ "demo.js", "demo.css" ]
        }
    },
    "pkg" : {}
}
```

在js中声明依赖

支持识别 js 文件中的 require 函数，或者 注释中的 @require 字段 标记的依赖关系，这些分析处理对 html 的 script 标签内容 同样有效。

```
//demo.js
/**
 * @require demo.css
 * @require list.js
 */
var $ = require('jquery');
```

在SourceMap中则可看到：

```
{
    "res" : {
        ...
        "demo.js" : {
            "uri" : "/static/js/demo_33c5143.js",
            "type" : "js",
            "deps" : [ "demo.css", "list.js", "jquery" ]
        },
```

```
        ...
    },
    "pkg" : {}
}
```

在css中声明依赖

支持识别 css 文件 注释中的 @require 字段 标记的依赖关系，这些分析处理对 html 的 style 标签内容 同样有效。

```
//demo.js
/**
 * @require demo.css
 * @require list.js
 */
var $ = require('jquery');
```

在SourceMap中则可看到：

```
{
    "res" : {
        ...
        "demo.js" : {
            "uri" : "/static/js/demo_33c5143.js",
            "type" : "js",
            "deps" : [ "demo.css", "list.js", "jquery" ]
        },
        ...
    },
    "pkg" : {}
}
```

按需加载静态资源

在静态资源管理系统接管了项目中的静态资源后，可以知道静态资源的运行情况以及依赖关系，然后可以做到自动为页面按需加载静态资源，下面通过一个例子来详细讲解：

sidebar.tpl 中的内容如下，

```
<!--
    @require "common:ui/dialog/dialog.css"
-->

<a id="btn-navbar" class="btn-navbar">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</a>

{script}
```

```
    var sidebar = require("common:ui/dialog/dialog.js");
    sidebar.run();
{/script}

{script}
    $('a.btn-navbar').click(function() {
        require.async('common:ui/dialog/dialog.async.js', function( dialog ) {
            dialog.run();
        });
    });
{/script}
```

对项目编译后，自动化工具会分析依赖关系，并生成 sourcemap，如下

```
"common:widget/sidebar/sidebar.tpl": {
    "uri": "common/widget/sidebsr/sidebar.tpl",
    "type": "tpl",
    "extras": {
        "async": [
            "common:ui/dialog/dialog.async.js"
        ]
    },
    "deps": [
        "common:ui/dialog/dialog.js",
        "common:ui/dialog/dialog.css"
    ]
}
```

在 sidebar 模块被调用后，静态资源管理系统通过查询 sourcemap 可以得知，当前 sidebar 模块同步依赖 sidebar.js、sidebar.css，异步依赖 sdebar.async.js，在要输出的 html 前面，生成静态资源外链，我们得到最终的 html

```
<link rel="stylesheet" href="/static/ui/dialog/dialog_7defa41.css">

<a id="btn-navbar" class="btn-navbar">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
</a>

<script type="text/javascript"
src="/static/common/ui/dialog/dialog$12cd4.js"></script>
<script type="text/javascript">
    require.resourceMap({
        "res": {
            "common:ui/dialog/dialog.async.js": {
                "url": "/satic/common/ui/dialog/dialog.async_449e169.js"
            }
        }
```

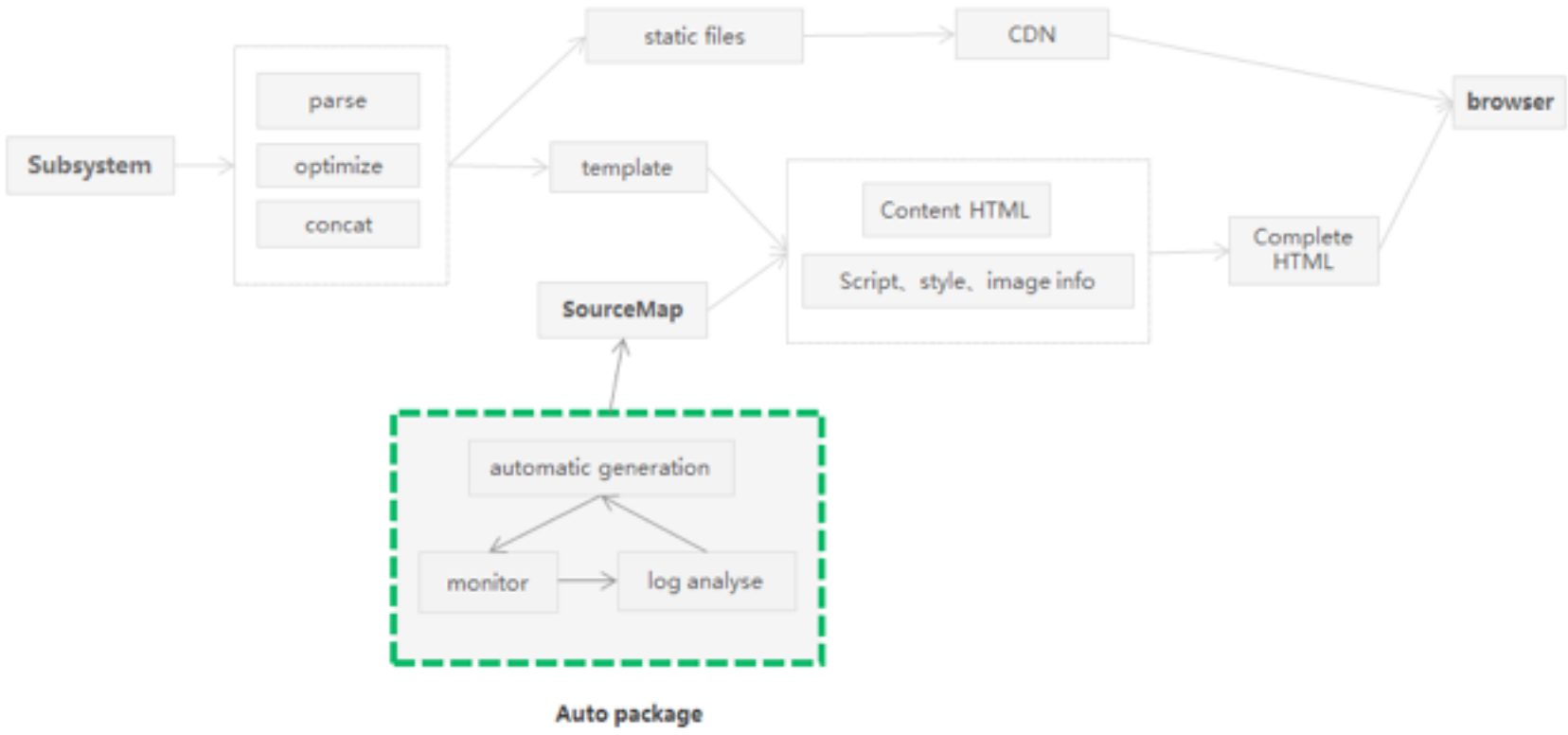
```
    });
</script>
<script type="text/javascript">
    var sidebar = require("common:ui/dialog/dialog.js");
    sidebar.run();

    $('a.btn-navbar').click(function() {
        require.async('common:ui/dialog/dialog.async.js', function( dialog ) {
            dialog.run();
        });
    });
</script>
```

如上可见，后端模块化框架将同步模块的 script url 统一生成到页面底部，将 css url 统一生成在 head 中，对于异步模块(require.async)注册 resourceMap 代码，框架会通过 {script} 标签收集到页面所有 script，统一管理并按顺序输出 script 到相应位置。

当我们想对模块进行打包，只需要使用一个 pack 配置项，对网站的静态资源进行打包，这样在 SourceMap 中，所有被打包的资源会有一个 pkg 属性指向该表中的资源，而这个资源，正是我们配置的打包策略。这样静态资源系统可以根据对应信息找到某个资源最终被合并后的 package 的 url，最后把这个 url 返回给页面。

自动合并静态资源

静态资源管理系统可以根据产品线上静态资源使用的数据，自动完成静态资源合并工作，对工程师完全透明，解决手工维护的未及时排除废弃资源、不可持续、成本大等问题。



详情请见 静态资源自动合并;

静态资源版本更新与缓存

静态资源管理系统采用基于文件内容的 hash 值来控制静态资源的版本更新，如下所示：

```
<script type="text/javascript" src="a_8244e91.js"></script>
```

其中”_82244e91 ”这串字符是根据 a.js 的文件内容进行 hash 运算得到的，只有文件内容发生变化了才会有更改。这样做的好处有：

- 线上的 a.js 不是同名文件覆盖，而是文件名 +hash 的冗余，所以可以先上线静态资源，再上线 html 页面，不存在间隙问题；
- 遇到问题回滚版本的时候，无需回滚 a.js，只须回滚页面即可；
- 由于静态资源版本号是文件内容的 hash，因此所有静态资源可以开启永久强缓存，只有更新了内容的文件才会缓存失效，缓存利用率大增；
- 修改静态资源后会在线上产生新的文件，一个文件对应一个版本，因此不会受到构造 CDN 缓存形式的攻击

静态资源管理系统会在 compile 阶段识别文件中的定位标记(url)，计算对应文件的 hash，并自动替换为 '文件名 + hash'，无需工程师手动修改。

静态资源分级控制

静态资源管理系统可以对静态资源做进一步控制(Controlling Access to Features)以达到分级发布的效果，主要包括以下两块核心功能，

- feature flags, 用来控制 feature 对应的静态资源是否加载
- feature flippers, 可以灵活控制 feature，不仅仅是 on 或 off, 可以做到类似'3%用户可以访问此功能'、'对内部所有员工开放' 类似的效果

通过以上的控制我们可以轻松做到发布一个新功能，让这个功能只对部分用户可访问，当功能完善后对所有用户开放，如果功能出现问题直接一键回滚即可。

在项目中的类似代码如下：

```
{if $config.some eq 'Fred'}
    do something new and amazing here.
{elseif $config.some eq 'Wilma'}
    do the current boring stuff.
{else}
    whatever you are.
```

静态资源管理系统会根据配置在运行时对 $config.some 进行干预.实现对静态资源的访问权控制，通过运行时的配置(feature flag)来控制静态资源，还可以支持“主干开发”的方式，来达到更快的迭代速度。

我们还可以实现国际化的需求，原理同分级发布，在运行时的做一些更细致的差异化处理

```
{if $lang == 'zh-CN'}
    zh-CN
{/if}
```

总结

静态资源管理系统的核心是对静态资源进行调度，可以很灵活的适应各种性能优化和差异化处理的场景，来达到更快、更可靠、低成本的自动化项目交付。但是同时这个系统十分复杂，承载着各种职责，这个系统本身会成为整个网站的关键节点和瓶颈。

作者：walter

原文链接：http://fex.baidu.com/blog/2014/04/fis-static-resource-management/

前端工程师必备技能汇总

项目起源

还记得@jayli 的这幅前端知识结构图么。

图片的形式具有诸多的不便。缺失源图的我们，无法为此图贡献些什么，随着时间的迁移，或许有些技术点会发生改变，所以有了这个github项目。我们可以通过协作的方式来共同维护这个项目。Git的历史记录也可以见证前端行业的一些变迁。

尽管会变成文字的方式来维护这些内容，但是我承诺写一个小工具帮大家生成更好玩的图形（基于DataV项目）。

前端开发知识结构

- 前端工程师
  - SVG/Canvas/VML
  - SVG: D3/Raphaël/Snap.svg/DataV
  - Canvas: CreateJS/KineticJS
  - 知识管理/总结分享
  - 沟通技巧/团队协作
  - 需求管理/PM
  - 交互设计/可用性/可访问性知识
  - 编译原理
  - 计算机网络
  - 操作系统
  - 算法原理
  - 软件工程/软件测试原理
  - D2/WebRebuild
  - NodeParty/W3CTech/HTML5梦工厂
  - JSConf/沪JS(JSConf.cn)
  - QCon/Velocity/SDCC
  - JSConf/NodeConf
  - CSSConf

- YDN/YUIConf
- HybridApp
- HTML5/CSS3
- 响应式设计
- Zeptojs/iScroll
- V5/Sencha Touch
- PhoneGap
- jQuery Mobile
- CSRF/XSS
- ADsafe/Caja/Sandbox
- 类库模块化
- 业务逻辑模块化
- 文件加载
- 模块化预处理器
- CommonJS/AMD
- YUI3模块
- bower/component
- LABjs
- SeaJS/Require.js
- Browserify
- 压缩合并
- 文档输出
- 项目构建工具
- YUI Compressor
- Google Clousure Complier
- UglifyJS
- CleanCSS
- JSDoc

- Dox/Doxmate/Grunt-Doxmate
- make/Ant
- GYP
- Grunt
- Yeoman
- FIS
- Mod
- 数据结构
- OOP/AOP
- 原型链/作用域链
- 闭包
- 函数式编程
- 设计模式
- Javascript Tips
- JSPerf
- YSlow 35 rules
- PageSpeed
- HTTPWatch
- DynaTrace's Ajax
- 高性能JavaScript
- HTTP1.1
- ECMAScript3/5
- W3C/DOM/BOM/XHTML/XML/JSON/JSONP
- CommonJS Modules/AMD
- HTML5/CSS3
- jQuery/Underscore/Mootools/Prototype.js
- YUI3/Dojo/ExtJS/KISSY
- Backbone/KnockoutJS/Emberjs

- AngularJS
- Bootstrap
- Semantic UI
- Batarang
- Coding style
- 单元测试
- 自动化测试
- JSLint/JSHint
- CSSLint
- Markup Validation Service
- QUnit/Jasmine
- Mocha/Should/Chai/Expect
- WebDriver/Karma Runner/Sahi
- phantomjs
- IDE
- 调试工具
- 版本管理
- VIM/Sublime Text2
- Notepad++/EditPlus
- WebStorm
- Emacs EmacsWiki
- Brackets
- Firebug/Firecookie
- YSlow
- IEDeveloperToolbar/IETester
- Fiddler
- Chrome Dev Tools
- Git/SVN

- Github/Bitbucket/Google Code
- HTML/HTML5
- CSS/CSS3
- PhotoShop/Paint.net/Fireworks
- JavaScript/Node.js
- CoffeeScript
- TypeScript
- IE6/7/8/9/10/11
- Firefox
- Chrome/Safari/Opera
- 浏览器
- 编程语言
- 切页面
- 开发工具
    - 代码质量
    - 前端库/框架
    - 前端标准/规范
    - 性能
    - 编程知识储备
    - 部署流程
    - 代码组织
    - 安全
    - 移动Web
    - 前沿技术社区/会议
    - 计算机知识储备
    - 软技能
    - 可视化
- 后端工程师

- Unix/Linux/OS X/Windows
- Varnish
- Squid
- Redis
- Memcached
- SQL
- MySQL/PostgreSQL/Oracle
- MongoDB/CouchDB
- Nginx
- Apache
- C/C++/Java/PHP/Ruby/Python/...
- 编程语言
- 服务器
- 数据库
    - 数据缓存
    - 文件缓存/代理
    - 操作系统
    - 数据结构

前端书籍推荐

★越少越简单,越适合入门,★多的要么是难度比较高,要么是比较适合在后面看,比如讲性能之类的书.

CSS

- Eric Meyer 谈 CSS（卷二）★★★
- CSS权威指南 （第3版）★★
- 精通CSS★★★

JavaScript

- JavaScript DOM编程艺术 （第2版）★
- JavaScript高级程序设计（第3版）★★

- 锋利的jQuery★★
- 高性能JavaScript★★★
- JavaScript语言精粹★★★
- JavaScript权威指南★★★
- 编写可维护的JavaScript★★★
- JAVASCRIPT语言精髓与编程实践★★★
- Effective Javascript★★★
- Secrets of the JavaScript Ninja★★★
- JavaScript模式★★★
- JavaScript设计模式★★★★
- 基于MVC的JavaScript Web富应用开发★★★

版本控制工具

- 版本控制之道 (git)★★
- Git权威指南★★★★

后端书籍推荐

Linux管理

- Linux 系统管理技术手册
- 鸟哥的 Linux 私房菜
- Linux 101 Hacks
- UNIX Shell Scripting
- The Linux Command Line

Linux编程

- Linux程序设计
- Linux系统编程
- Unix环境高级编程
- Unix编程艺术
- The Linux Programming Interface

- 程序员的自我修养
- 深入理解Linux内核
- Unix网络编程
- TCP/IP高级编程

C/C++

- Linux C编程一站式学习
- C和指针
- C陷阱与缺陷
- C专家编程
- C语言核心技术
- 彻底搞定C指针
- 征服C指针
- C++编程思想
- 高质量程序设计指南---C/C++语言
- Inside the C++ Object Model

内容贡献者

除了感谢Jayli提供了知识结构图的原本来，还感谢以下的内容贡献者们，结果由git-summary生成于2014-01-03：

```
project  : fks
repo age : 1 year, 3 months
active   : 53 days
commits  : 108
files    : 4
authors  :
   56  Jackson Tian             51.9%
    9  吴晓兰                 8.3%
    5  liyinkan                 4.6%
    3  chriscai                 2.8%
    3  fengxiaolong             2.8%
    3  XiNGRZ                   2.8%
    2  monkadd                  1.9%
    2  Johnny                   1.9%
    2  weiwenqing               1.9%
    2  Yinkan Li                1.9%
```

```
2  Copypeng                1.9%
2  左岸                  1.9%
2  Jakukyo Friel           1.9%
2  Glowin                  1.9%
1  李亚川                0.9%
1  Evan You                0.9%
1  Mickey                  0.9%
1  Mickey-                 0.9%
1  Qi Junyuan              0.9%
1  browsnet                0.9%
1  doabit                  0.9%
1  guoxiangyang            0.9%
1  linkgod                 0.9%
1  popomore                0.9%
1  vipzhicheng             0.9%
1  zhaqiang                0.9%
1  Colin Luo               0.9%
```

文章转载自：开源中国社区 [http://www.oschina.net]

本文地址：http://www.oschina.net/news/50654/frontend-knowledge-structure

原文地址: https://github.com/JacksonTian/fks

C++11新特性：自动类型推断和类型获取

作者/PumpkinDylan
面对现实，忠于理想

自动类型推断

当编译器能够在一个变量的声明时候就推断出它的类型，那么你就能够用auto关键字来作为他们的类型：

```
auto x = 1;
```

编译器当然知道x是integer类型的。所以你就不用int了。接触过泛型编程或者API编程的人大概可以猜出自动类型推断是做什么用的了：帮你省去大量冗长的类型声明语句。

比如下面这个例子：

在原来的C++中，你要想使用vector的迭代器得这么写：

```
vector<int> vec;
vector<int>::iterator itr = vec.iterator();
```

看起来就很不爽。现在你可以这么写了：

```
vector<int> vec;
auto itr = vec.iterator();
```

果断简洁多了吧。假如说自动类型推断只有这样的用法的话那未免也太naive了。在很多情况下它还能提供更深层次的便利。

比如说有这样的代码：

```
template <typename BuiltType, typename Builder>
void makeAndProcessObject (const Builder& builder)
{
    BuiltType val = builder.makeObject();
    // do stuff with val
}
```

这个函数的功能是要使用builder的makeObject产生的实例来进行某些操作。但是现在引入了泛型编程。builder的类型不同，那么 makeObject返回的类型也不同，那么我们这里就得引入两个泛型。看起来很复杂是吧，所以这里就可以使用自动类型推断来简化操作：

```
template <typename Builder>
void makeAndProcessObject (const Builder& builder)
{
    auto val = builder.makeObject();
    // do stuff with val
}
```

因为在得之builder的类型之后，编译器就已经能知道makeObject的返回值类型了。所以我们能够让编译器自动去推断val的类型。这样一来就省去了一个泛型。

你以为自动类型推断只有这样的用法？那也太naive了。C++11还允许对函数的返回值进行类型推断

新的返回值语法和类型获取（DecItype)语句

在原来，我们声明一个函数都是这样的：

```
int temp(int a, double b);
```
前面那个int是函数的返回值类型，temp是函数名，int a, double b是参数列表。

现在你可以将函数返回值类型移到到参数列表之后来定义：

```
auto temp(int a, double b) -> int;
```

后置返回值类型可以有很多用处。比如有下列的类定义：

```
class Person
{
public:
    enum PersonType { ADULT, CHILD, SENIOR };
    void setPersonType (PersonType person_type);
    PersonType getPersonType ();
private:
    PersonType _person_type;
};
```

那么在定义getPersonType函数的时候我们得这么写：

```
Person::PersonType Person::getPersonType ()
{
    return _person_type;
}
```

因为函数所在的类Person是声明在函数返回值之后的，所以在写返回值的时候编译器并不知道这个函数是在哪个类里面。由于PersonTYpe是 Person类的内部声明的枚举，所以在看到PersonType的时候，编译器是找不到这个类型的。所以你就得在PersonTYpe前面加上Person：：，告诉编译器这个类型是属于Person的。这看起来有点麻烦是吧。当你使用新的返回值语法的时候呢就可以这么写：

```
auto Person::getPersonType () -> PersonType
{
    return _person_type;
}
```

因为这次编译器看到返回值类型PersonType的时候已经知道这个函数属于类Person。所以它会到Person类中去找到这个枚举类型。

当然上述应用只能说是一个奇技淫巧而已。并没有帮我们多大的忙（代码甚至都没有变短）。所以还得引入C++11的另一个功能。

类型获取（Decltype）

既然编译器能够推断一个变量的类型，那么我们在代码中就应该能显示地获得一个变量的类型信息。所以C++介绍了另一个功能：decltype。（实在是不知道这个词该怎么翻译，姑且称之为类型获取）。

```
int x = 3;
decltype(x) y = x; // same thing as auto y = x;
```

上述代码就使用了类型获取功能。和函数返回值后置语法结合起来，可以有如下应用：

```
template <typename Builder>
auto
makeAndProcessObject (const Builder& builder) -> decltype( builder.makeObject()
)
{
    auto val = builder.makeObject();
    // do stuff with val
    return val;
}
```

前面的例子中这个函数的返回值是void，所以不需要为返回值引入泛型。如果返回值是makeObject的返回值的话，那么这个函数就得引入两个 泛型。现在又了类型获取功能，我们就能在返回值中自动推断makeObject的类型了。所以decltype确实为我们提供了很大的便利。

这个功能非常重要，在很多时候，尤其是引入了泛型编程的时候，你可能记不住一个变量的类型或者类型太过复杂以至于写不出来。你就要灵活使用decltype来获取这个变量的类型。

作者：@pumpkindylan

原文链接：http://blog.csdn.net/srzhz/article/details/7934483

文章转载自：http://blog.segmentfault.com/pumpkindylan/1190000000460962

JAVA 8:ORM已经过时了

译者/Java译站
专注IT技术文章翻译分享
http://it.deepinmind.com

ORM已经过时了

最近几十年来，关于ORM究竟还有没有用的争论一直不断。很多人承认Hibernate和JPA确实很好的解决了不少实际的问题（通常是复杂对象的持久化），但有些人认为，对于面向数据的应用而言，复杂的映射关系则有点大材小用了。

JPA通过在目标类型上使用硬编码的注解，来建立标准的声明式的映射规则，进而完成映射关系。但我们认为，很多以数据为中心的应用不应该受限于注 解的局限性，而应该通过一种函数式的方式来解决。Java 8的Steam API终于让我们可以用一种简洁的方式来解决这个问题了！

我们先从一个简单的例子开始，这里我们使用H2的INFORMATION_SCHEMA来查询所有的表及字段。我们专门用一个Map<String, List<String>>类型的数据结构来存储这个信息。我们用jOOQ来简化SQL的交互。下面来做一下准备工作：

Java代码

```
1.  public static void main(String[] args)
2.  throws Exception {
3.      Class.forName("org.h2.Driver");
4.      try (Connection c = getConnection(
5.              "jdbc:h2:~/sql-goodies-with-mapping
    ",
6.              "sa", "")) {
7.
8.          //
     This SQL statement produces all table
9.          //
     names and column names in the H2 schema
10.         String sql =
11.             "select table_name, column_name " +

12.             "from information_schema.columns "
    +
13.             "order by " +
14.                 "table_catalog, " +
15.                 "table_schema, " +
16.                 "table_name, " +
17.                 "ordinal_position";
18.
19.         //
     This is jOOQ's way of executing the above
20.         // statement. Result implements List, w
    hich
21.         // makes subsequent steps much easier
```

```
22.          Result<Record> result =
23.          DSL.using(c)
24.             .fetch(sql)
25.     }
26. }
```

我们已经执行完查询了，现在来看下如何从查询结果中生成Map<String, List<String>>信息。

Java代码

```
1.  DSL.using(c)
2.     .fetch(sql)
3.     .stream()
4.     .collect(groupingBy(
5.         r -> r.getValue("TABLE_NAME"),
6.         mapping(
7.             r -> r.getValue("COLUMN_NAME"),
8.             toList()
9.         )
10.    ))
11.    .forEach(
12.        (table, columns) ->
13.            System.out.println(table + ": " + columns)
14.    );
```

上述程序会输出如下的结果：

Java代码

1. FUNCTION_COLUMNS: [ALIAS_CATALOG, ALIAS_SCHEMA, ...]
2. CONSTANTS: [CONSTANT_CATALOG, CONSTANT_SCHEMA, ...]
3. SEQUENCES: [SEQUENCE_CATALOG, SEQUENCE_SCHEMA, ...]

这是如何工作的？我们来一步一步的分析下：

Java代码

```
1.  DSL.using(c)
2.     .fetch(sql)
3.  //这里我们把一个列表转化成一个Stream
4.   .stream()
5.  // 把Stream中的元素汇集成一个新的类型
6.     .collect(
7.  // 收集器是一个分组操作，会生成一个map
8.                groupingBy(
9.  // 分组操作的key是jOOQ记录的TABLE_NAME字段
10.         r -> r.getValue("TABLE_NAME"),
11. // 分组操作的值是用这个映射表达式来生成的
```

```
        mapping(
// 实际上是映射到每条jOOQ记录的COLUMN_NAME字段上
            r -> r.getValue("COLUMN_NAME"),
// 然后将所有的值收集到一个java.util.List中
            toList()
        )
    ))
// 一旦拿到了<String, List<String>>列表，
// 就可以很容易通过lambda表达式来使用它了
    .forEach(
        (table, columns) ->
            System.out.println(table + ": " + columns)
    );
```

看明白了吗？第一次使用这些方法的话可能的确需要费点工夫。新的类型，泛型，lambda表达式，这些东西加到一起的话，第一次看到的话的确会有 些困惑。最好的办法就是不断的去实践直到你完全掌握了。毕竟来说，和Java Collections API比起来的话，Stream API可是一次革命性的进步。

不过有个好消息就是，这些方法已经确定就是这样，不会再变了。你的每一次实践都是对未来的一次投资。

注意上述的程序用到了如下的静态导入语句：

Java代码

```
import static java.util.stream.Collectors.*;
```

同样还应该注意到，输出的结果和数据库返回的顺序是不一样的。这是因为groupingBy收集器返回的是一个java.util.HashMap。在这个例子中，我们更希望能收集到一个java.util.LinkedHashMap里面，这个集合能保留插入时的顺序。

Java代码

```
DSL.using(c)
    .fetch(sql)
    .stream()
    .collect(groupingBy(
        r -> r.getValue("TABLE_NAME"),

        // Add this Supplier to the groupingBy
        // method call
        LinkedHashMap::new,
        mapping(
            r -> r.getValue("COLUMN_NAME"),
            toList()
        )
    ))
```

```
15.     .forEach(...);
```

我们还可以用另一种转化结果的方法。想像一下，我们将要从上面的schema中生成DDL。这很简单。首先，我们需要知道每一列的数据类型。只需要把它加到我们的SQL查询语句中就好了：

Java代码

```
1.  String sql =
2.      "select " +
3.          "table_name, " +
4.          "column_name, " +
5.          "type_name " + // Add the column type
6.      "from information_schema.columns " +
7.      "order by " +
8.          "table_catalog, " +
9.          "table_schema, " +
10.         "table_name, " +
11.         "ordinal_position";
```

这个例子中还引入了一个新的局部类，用来封装名字和类型属性：

Java代码

```
1.  class Column {
2.      final String name;
3.      final String type
4.      Column(String name, String type) {
5.          this.name = name;
6.          this.type = type;
7.      }
8.  }
```

现在我们来看下Streams API的方法调用该如何修改：

Java代码

```
1.  result
2.      .stream()
3.      .collect(groupingBy(
4.          r -> r.getValue("TABLE_NAME"),
5.          LinkedHashMap::new,
6.          mapping(
7.
8.              // We now collect this new wrapper type
9.              // instead of just the COLUMN_NAME
10.             r -> new Column(
11.                 r.getValue("COLUMN_NAME", String.class),
12.                 r.getValue("TYPE_NAME", String.class)
```

```
            ),
            toList()
        )
    ))
    .forEach(
        (table, columns) -> {

            // Just emit a CREATE TABLE statement
            System.out.println(
                "CREATE TABLE " + table + " (");

            // Map each "Column" type into a String
            // containing the column specification,
            // and join them using comma and
            // newline. Done!
            System.out.println(
                columns.stream()
                       .map(col -> "  " + col.name +
                                   " " + col.type)
                       .collect(Collectors.joining(",\n"))
            );

            System.out.println(");");
        }
    );
```

输出的结果简直是棒极了!

Java代码

```
CREATE TABLE CATALOGS(
  CATALOG_NAME VARCHAR
);
CREATE TABLE COLLATIONS(
  NAME VARCHAR,
  KEY VARCHAR
);
CREATE TABLE COLUMNS(
  TABLE_CATALOG VARCHAR,
  TABLE_SCHEMA VARCHAR,
  TABLE_NAME VARCHAR,
  COLUMN_NAME VARCHAR,
  ORDINAL_POSITION INTEGER,
  COLUMN_DEFAULT VARCHAR,
  IS_NULLABLE VARCHAR,
  DATA_TYPE INTEGER,
  CHARACTER_MAXIMUM_LENGTH INTEGER,
  CHARACTER_OCTET_LENGTH INTEGER,
  NUMERIC_PRECISION INTEGER,
  NUMERIC_PRECISION_RADIX INTEGER,
```

```
    NUMERIC_SCALE INTEGER,
    CHARACTER_SET_NAME VARCHAR,
    COLLATION_NAME VARCHAR,
    TYPE_NAME VARCHAR,
    NULLABLE INTEGER,
    IS_COMPUTED BOOLEAN,
    SELECTIVITY INTEGER,
    CHECK_CONSTRAINT VARCHAR,
    SEQUENCE_NAME VARCHAR,
    REMARKS VARCHAR,
    SOURCE_DATA_TYPE SMALLINT
);
```

激动吧？看来ORM的时代已经过去了

ORM的时代应该终结了。为什么？因为软件工程里有一个很重要的思想就是使用函数式表达式来进行数据集的转化。函数式编程表达性强，并且非常通 用。它是数据及数据流处理的核心。Java开发人员现在也都知道函数式编程，而大家又都用过SQL。相像一下吧。你用SQL来声明表来源，把数据转化成新 的元组流，然后要么将它们作为派生表提供给其它更高级的SQL语句来使用，要么将它们交给你的应用程序来处理。

如果你用的是XML的话，你可以使用XSLT来声明XML转化，并通过XProc管道把结果提供给其它XML处理器，比如说另一个XSL表格。

使用SQL和Streams API是数据处理里面一个很重要的概念。如果你使用了jOOQ的话，你还可以进行类型安全的数据库访问及查询。想像一下如何使用jOOQ的流API来改写前面的代码，而不是直接使用SQL语句。

整个方法调用链会是一个流式的数据转化链，就像这样：

Java代码

```
DSL.using(c)
    .select(
        COLUMNS.TABLE_NAME,
        COLUMNS.COLUMN_NAME,
        COLUMNS.TYPE_NAME
    )
    .from(COLUMNS)
    .orderBy(
        COLUMNS.TABLE_CATALOG,
        COLUMNS.TABLE_SCHEMA,
        COLUMNS.TABLE_NAME,
        COLUMNS.ORDINAL_POSITION
    )
    .fetch()  // jOOQ ends here
    .stream() // Streams start here
    .collect(groupingBy(
```

```
        r -> r.getValue(COLUMNS.TABLE_NAME),
        LinkedHashMap::new,
        mapping(
            r -> new Column(
                r.getValue(COLUMNS.COLUMN_NAME),
                r.getValue(COLUMNS.TYPE_NAME)
            ),
            toList()
        )
    ))
    .forEach(
        (table, columns) -> {
             // Just emit a CREATE TABLE statement
             System.out.println(
                 "CREATE TABLE " + table + " (");

             // Map each "Column" type into a String
             // containing the column specification,
             // and join them using comma and
             // newline. Done!
             System.out.println(
                 columns.stream()
                        .map(col -> "  " + col.name +
                                    " " + col.type)
                        .collect(Collectors.joining(",\n"))
                 );

             System.out.println(");");
         }
    );
```

Java 8代表着未来，而有了jOOQ,Java 8以及Streams API,你可以写出强大的数据转化的API。希望你能和我们一样感到激动！还有更多的Java 8的内容，敬请收看。

英文原文：http://blog.jooq.org/2014/04/11/java-8-friday-no-more-need-for-orms/
译者：@Java译站
本文转载自：http://deepinmind.iteye.com/blog/2044307

使用CAS实现无锁的SKIPLIST

作者/付哲
阿里云开发工程师

无锁

并发环境下最常用的同步手段是互斥锁和读写锁，例如pthread_mutex和pthread_readwrite_lock，常用的范式为：

```
void ConcurrencyOperation() {
    mutex.lock();
    // do something
    mutex.unlock();
}
```

这种方法的优点是：

1. 编程模型简单，如果小心控制上锁顺序，一般来说不会有死锁的问题；
2. 可以通过调节锁的粒度来调节性能。

缺点是：

1. 所有基于锁的算法都有死锁的可能；
2. 上锁和解锁时进程要从用户态切换到内核态，并可能伴随有线程的调度、上下文切换等，开销比较重；
3. 对共享数据的读与写之间会有互斥。

无锁编程（严格来讲是非阻塞编程）可以分为lock free和wait-free两种，下面是对它们的简单描述：

- lock free：锁无关，一个锁无关的程序能够确保它所有线程中至少有一个能够继续往下执行。这意味着有些线程可能会被任意的延迟，然而在每一个步骤中至少有一个线程能够执行下去。因此这个系统作为一个整体总是在前进的，尽管有些线程的进度可能没有其它线程走的快。
- wait free：等待无关，一个等待无关的程序可以在有限步之内结束，而不管其它线程的相对执行速度如何。
- lock based：基于锁，基于锁的程序无法提供上面的任何保证，任一线程持有了某互斥体并处于等待状态，那么其它想要获取同意互斥体的线程只有等待，所有基于锁的算法无法摆脱死锁的阴影。

本文提到的无锁单指lock free。

lock free与CAS

常见的lock free编程一般是基于CAS(Compare And Swap)操作：

CAS(void *ptr, Any oldValue, Any newValue);
即查看内存地址ptr处的值，如果为oldValue则将其改为newValue，并返回true，否则返回false。X86平台上的CAS操作一般是通过CPU的CMPXCHG指令来完成的。CPU在执行此指令时 会首先锁住CPU总线，禁止其它核心对内存的访问，

然后再查看或修改*ptr的值。简单的说CAS利用了CPU的硬件锁来实现对共享资源的串行使用。它的 优点是：

1. 开销较小：不需要进入内核，不需要切换线程；
2. 没有死锁：总线锁最长持续为一次read+write的时间；
3. 只有写操作需要使用CAS，读操作与串行代码完全相同，可实现读写不互斥。

缺点是：

1. 编程非常复杂，两行代码之间可能发生任何事，很多常识性的假设都不成立。
2. CAS模型覆盖的情况非常少，无法用CAS实现原子的复数操作。

而在性能层面上，CAS与mutex/readwrite lock各有千秋，简述如下：

1. 单线程下CAS的开销大约为10次加法操作，mutex的上锁+解锁大约为20次加法操作，而readwrite lock的开销则更大一些。
2. CAS的性能为固定值，而mutex则可以通过改变临界区的大小来调节性能；
3. 如果临界区中真正的修改操作只占一小部分，那么用CAS可以获得更大的并发度。
4. 多核CPU中线程调度成本较高，此时更适合用CAS。

使用CAS实现无锁单向链表

单向链表实现的核心就是insert函数，这里我们用两个版本的insert函数来进行简单的演示，使用的CAS操作为GCC提供的__sync_compare_and_swap函数。

首先是无序的insert操作，即将新结点插入到指定结点的后面。

```
void insert(Node *prev, Node *node) {
    while (true) {
        node->next = prev->next;
        if (__sync_compare_and_swap(&prev->next, node->next, node)) {
            return;
        }
    }
}
```

代码分析：

1. 首先修改node->next，此时node还没有完成插入，只能被本线程看到，因此这个修改可以直接进行。
2. 在if中尝试修改prev->next，如果失败，则表明prev->next刚刚被其它线程修改了，则重复这一过程。

然后是有序的insert操作，即保证prev<= node <= next。

```
void insert(Node *prev, Node *node) {
    while (true) {
        Node *next = prev->next;
        while (next != NULL && next->item < node->item) {
            prev = next;
            next = prev->next;
        }
        node->next = next;
        if (__sync_compare_and_swap(&prev->next, next, node)) {
```

```
            return;
        }
    }
}
```

这段代码相比上一版本多了一个next变量。如果去掉next变量，那么代码就是下面的样子。

```
void insert(Node *prev, Node *node) {
    while (true) {
        while (prev->next != NULL && prev->next->item < node->item) {
            prev = prev->next;
        }
        node->next = prev->next;
        if (__sync_compare_and_swap(&prev->next, node->next, node)) {
            return;
        }
    }
}
```

上面的代码有着很严重的安全隐患：prev是共享资源，因此每个prev->next的值不一定是相等的！解决办法就是用一个局部变量来保存某个时刻prev的值，从而保证我们在不同地方进行比较的结点是一致的。

Key-Value数据结构

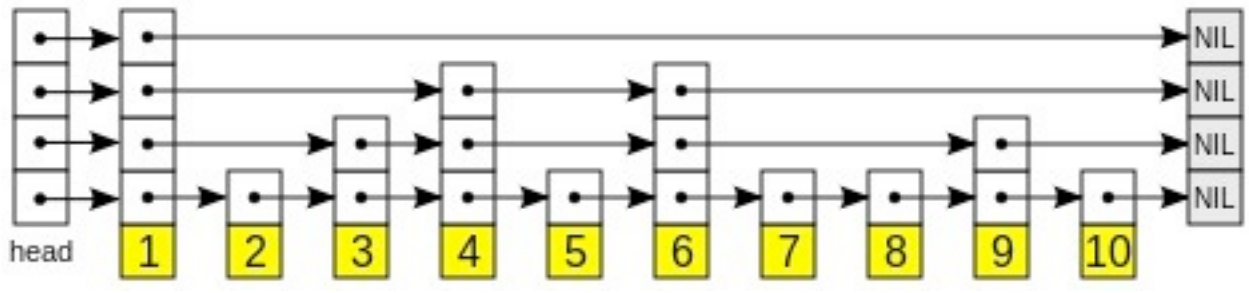
目前常用的key-value数据结构有三种：Hash表、红黑树、SkipList，它们各自有着不同的优缺点（不考虑删除操作）：

1. Hash表：插入、查找最快，为O(1)；如使用链表实现则可实现无锁；数据有序化需要显式的排序操作。
2. 红黑树：插入、查找为O(logn)，但常数项较小；无锁实现的复杂性很高，一般需要加锁；数据天然有序。
3. SkipList：插入、查找为O(logn)，但常数项比红黑树要大；底层结构为链表，可无锁实现；数据天然有序。

如果要实现一个key-value结构，需求的功能有插入、查找、迭代、修改， 那么首先Hash表就不是很适合了，因为迭代的时间复杂度比较高；而红黑树的插入很可能会涉及多个结点的旋转、变色操作，因此需要在外层加锁，这无形中降 低了它可能的并发度。而SkipList底层是用链表实现的，可以实现为lock free，同时它还有着不错的性能（单线程下只比红黑树略慢），非常适合用来实现我们需求的那种key-value结构。LevelDB、Reddis的 底层存储结构就是用的SkipList。

SkipList

那么，SkipList是什么呢？它由多层有序链表组成，每层链表的结点数量都是上一层的X倍，而它的插入和查找操作都从顶层开始进行。



（图片取自wiki）

从上图可以很容易看出查找的方式：

1. 从顶层的头结点出发；
2. 若下一结点为目标值，则返回结果；
3. 若下一结点小于目标值，则前进；
4. 若下一结点大于目标值或为NULL，则：
    1. 若当前处于最底层，则返回NULL；
    2. 下降一层，重复2-4步。

在SkipList中，结点层数非常关键，如果各个结点的层数均匀分布，那么插入与查找的效率就会比较高。为了实现这一目的，SkipList中每个结点的层数是在插入前随机算出来的，其基本原理就是令结点在i层的概率是i+1层的X倍，代码如下：

```
int RandLevel(int X, int maxLevel) {
    int r = rand();
    int level = 1;
    for (int j = X; r < RAND_MAX / j && level < maxLevel; ++level, j *= X)
        continue;
    return level;
}
```

插入新结点的过程与查找很类似，这里我们假设链表中的各结点不允许重复：

1. 计算出新结点的层数lv；
2. 从lv层的头结点出发，开始查找过程；
3. 如果找到目标值，返回NULL；
4. 如果当前处于最底层，则创建新结点，并依次将新结点插入到1-lv层；

可以看出，插入操作的1-3步是单纯的读操作，只有第4步才是对共享资源的写操 作。而第4步的插入实质上就是有序链表的插入操作，我们在前面已经简述了如何用CAS实现它。因此，只要保证插入顺序是从底层向上依次插入，那么就可以将 SkipList实现为lock free。插入顺序从底向上进行的原因如下。

N个插入操作肯定需要至少N次CAS，而任意一个CAS成功后就意味着新结点已经成为了SkipList的一部分，变成了共享资源，则新结点就需要遵循其它结点的原则：每个结点都同时存在于1-lv层。容易看出，只有从底层向上插入才能满足这一条件。

多个CAS操作本身没有原子性，即在N次插入没有完成前，新结点会表现出一定的不一致性，具体来说就是多个线程先后访问新结点时，看到的它的层数并不相同。这种不一致性会比较轻微的影响SkipList的性能，而不会影响它的正确性。

SkipList的插入代码如下：

```
void Insert(Node *node) {
    node->level = RandLevel(2, MAX_LEVEL);
    InsertInternal(head, node->level, node);
}
Node *InsertInternal(Node *prev, int lv, Node *node) {
    Node *next = prev->next[lv];
    while (next != NULL && next->item < node->item) {
        prev = next;
        next = prev->next[lv];
    }
    if (next == NULL || next->item > node->item) {
        if (lv != 0) {
            if (InsertInternal(prev, lv - 1, node) != NULL) {
                ListInsert(prev, node, lv);
            }
        }
    } else if (next->item == node->item) {
        return NULL;
    }
    return node;
}
```

其中ListInsert就是对前面有序链表插入的一个简单改写。整个插入过程递归实现，从而满足了插入顺序要从底向上的要求。

更多思考

在设计无锁SkipList时，不光需要我们将显式的锁用CAS替换掉，还需要尽量避免一些隐式的锁，以及一些非线程安全的函数。

1. RandLevel中的rand()是非线程安全的函数，需要替换为线程安全的版本（如非标准库的rand_r()），或是由各线程自己来保存rand使用的seed。
2. 在创建SkipList的时候需要指定一个MAX_LEVEL，即头结点的层数，这个值在此SkipList生命期中固定不变。一般来说12-20层都是可以接受的。
3. 全局new内部会加锁，如果这里有瓶颈的话需要换用自定义的内存池。
4. 如果使用了内存池，那么必须确保内存池本身是无锁且支持并发写的。否则就只能将SkipList改写为单写多读版本。
5. 在计算新结点的层数时，需要传入一个maxLevel，这里有两种常见做法：可以传入SkipList的最大层数MAX_LEVEL，也可以传入当前最大层数topLevel + 1。两种做法的优缺点为：
    1. 传入MAX_LEVEL可能在SkipList中结点数量较少时就达到很高的层数，降低了此时插入与查找的性能；但如果有序插入多个新结点，能保证各结点的层数均匀分布。
    2. 传入topLevel + 1可以保证在结点数较少时不太可能出现很高的层数，但在有序插入多个新结点时，可能导致前面插入结点的层数整体要低于后面插入的结点。
6. SkipList的修改操作也需要是lock free的，因此需要将Node中的item改为指针，在修改某结点值的时候用CAS来替换掉旧指针，并在完成后删除。
7. SkipList也可以在最底层加入反向指针prev，这样就能直接O(1)的反向迭代。带来的问题是更大的不一致性——在插入未完成时两个线程分别正向和反向迭代，看到的SkipList是不一致的。但可以保证SkipList在插入完成后的最终状态是一致的。

本文只是对无锁SkipList设计的一个简单回顾，不包括详细的实现代码。因为还不确定自己设计的有没有纰漏，还需要认真学习一下LevelDB和Reddis中的SkipList代码。

参考：

http://en.wikipedia.org/wiki/Skip_list

http://www.myexception.cn/ai/972131.html

http://www.seflerzhou.net/post-6.html

http://coolshell.cn/articles/8239.html

http://blog.csdn.net/sunmenggmail/article/details/12648465

作者：付哲

文章转载自：http://ifeve.com/cas-skiplist/

从零开始编写自己的C#框架（3）——开发规范

作者/Empty
.net码农
只有将自己置空，才能装进更多的东西！

由于是业余时间编写，而且为了保证质量，对写出来的东西也会反复斟酌，所以每周只能更新两章左右，请大家谅解，也请大家耐心等待，谢谢大家的支持。

初学者应该怎样学习本系列内容呢？根据我自己的学习经验，一般直接看一遍的方法，学习与认知都会比较浅，很快就忘了。而看完后写笔记、手抄或将所看的内容照着打一遍或多次的，可以比较深刻的理解文章或代码中的思想，并能将里面的核心内容牢记在心。

对于开发规范，都是老生常谈的事情了，很多正规一些的公司都有一套规范来约束，这些规范都是大同小异。

规范为什么那么重要？为什么大家反复提而初学者们都是当作耳边风，无视这些要求呢？下面先举几个例子给初学者说明一下。

08年的时候在一家主要做OA开发的公司，负责OA的二次开发，当时接到项目代码时傻眼了，没有文档、没有数据字典，代码规范就不要讲了，代码中的变量命名与数据库表和字段全是用拼音首字母来命名的，当时真有想死的感觉。当时只能感叹说，原开发人员真是太奇葩了。

记得几年前带过的一批应届生（当时公司对代码要求不是很严格，只追求能快速出 产品），在进公司的时候我也详细为他们讲了开发规范以及相关要求，然后进入开发状态。而对于这些新进公司的同事来说，开始时应该都会觉得我比较烦，又要求 开发规范也要求编写开发文档。不过因为公司不太重视，所以当时也没有很好的实行下去，那他们对开发规范也就不重视了。

一开始做的项目还好，给一些不太复杂的需求，而我也会经常检查他们的代码，每 周大概会有一到两天抽空开个小会议，为他们讲解代码中存在的问题以及一些算法，所以代码虽然不太规范，但也没有什么问题。后来有一段时间特忙没空理会大 家，而他们也开始各自独立负责一些小项目开发。其中一位开发的是一款《斗地主》KJava手机游戏，经过一段时间努

力奋战后游戏完成了，而这位同事由于个 人原因也同时请辞离职，可想而知这个项目是什么结果。项目在测试时发现存在不少BUG，想叫人接手这个项目时，才发现无人可以接手，因为项目里的注释量 少，又没有完善的开发文档，而代码编码也不够规范，花上不少时间研究也没弄清里面的关系.....最终的结果是公司花了不少投入而得出一个无法维护的项 目。

还有一个比较经典的案例也是KJava手机项目，做的是《日历》类的应用软 件，开发人员花了大量的时间，研究中国农历的时辰、二十八宿、五行、天干地支、民俗吉凶日等内容，终于开发出能自动计算每日时辰吉凶禁忌的手机日历，投入 到市场也有很不错的下载量和收入，推广效果很好。产品投入市场三个月后，领导要求对该项目进行二次开发，增加一些新的需求，当这位同事重新熟悉自己写的代 码时才发现，自己也很难看懂自己写过的代码了~~~大家是不是觉得不太可能发生这种事情，自己写过的代码也会不记得？呵呵.....主文件2K多行代码 没有多少注释，功能调用比较混乱，命名与编码也不规范，再加上各种时辰吉凶禁忌的计算.....听到这些你是否已经开始晕了，最后他自己也只能做一些 UI的修改，当然在这以后他对代码规范的重视也就完全不一样了。

当然例子还有很多，这类型的例子并不是偶然现象，相信类似的情型也曾发生过在不少公司项目中，为什么会发生这么多类似的案例呢？大家在追求高效开发，快速产出的时候，与代码规范也并不冲突啊?

接触多不少类似项目后，发现主要原因可能有这几个方面:

第一是没有一个好的开发框架，对于新人来说，还没有养成良好的开发习惯，接触 的项目也很少，他们不知道如何去规范自己的代码，也没有成熟的例子给他们模仿，在开发的时候过于自由没有约束的话，就会自由发挥，将那里影响自己开发速 度，不喜欢的事情全部剔除，然后做出的东西就非常个性化啦;

第二是不懂得开发规范的重要性（根本不重视），觉得自己写的代码自己肯定可以看得懂，至于其他人能否看得明白那就无所谓啦，看不明白就最好，那样的话公司就离不开自己，万一公司想炒掉自己这个项目可能就黄了。或者代码自己很熟悉，自以为别人也一样能看明白。但他们没有考虑的是，可读性好、严格执行开发规范的代码在应聘新公司时其实就是一张非常好的敲门砖。

第三是公司领导不重视，上行下效，如果领导层都不重视，那怎么将开发规范贯彻下去?

第四是技术主管不懂得代码规范以及其重要性。

.....

当然也有可能是其他方面的原因，但最重要的还是开发者自身问题，如果你有有一个良好的开发习惯，那么其他外因都影响不到你。

而遵守开发规范真的会浪费你的开发时间，降低开发效率吗?

对于我本人的开发习惯来说，代码中的注释非常多，几乎占代码量的1/3到1 /2（这只是个人习惯，并不推荐大家都用这种方法），很多人都会觉得不可思议，也许很多人都会想，你这个家伙太浪费时间了，老板真是浪费金钱养 你.....哈哈.....真正的事实是我所在公司的技术团队中，10年以上经验的大牛占了一半，而我的开发速度与效率可以排在前二，为什么呢？第一 我对开发规范比较重视，已成为我的一种习惯；第二我打字非常快；第三我对项目很了解；第四我必须这么做，因为开发出来的框架、封装好的类与函数是直接提供 给其他同事调用的，如果开发不够规范且注释又少的话，那其他同事就会很头痛了，那么可能要花很多时间与大家沟通说新增了什么功能，它的做什么用的，该怎么 调用等等，这样会浪费很多开发时间。

严格按开发规范实施，编写大量注释表面看会占用不少时间，但从长期的角度来 看，它提高了团队的开发效率，对二次开发维护也有非常重要的帮助。大家可以设想一下，几十万行的代码，如果没有好的注解，那就得天天在做猜迷游戏，经常要 与大家沟通，了解某个函数是如何使用的，或干脆不用别人已开发的功能，自己重新写过，那太多的个性化会使项目将来维护起来特别麻烦，这将很大的浪费团队开 发效率。而项目在进行二次开发维护时，由于时间跨度或接手的人不同了，那他要了解之前的功能以及相关业务流程，那将是多少苦逼的事情。

如果一个系统只需要100行代码，那里问题也不大。如果有一千行代码，那么认 真研究一下也没可以搞定。要是一万行、十万行以上代码时，不要说别人要理解你开发的框架中，各个接口、类、函数是干什么用的，就算是自己过了段时间也会不 清楚为什么代码是这样写的，设置某个参数到底有什么用，某些函数是做什么用的.....

看过有园友在博客上说，只要命名规范，写不写注释是无所谓的事情， 呃.....这个嘛要根据国情，国内很多开发人员的E文并不怎么样，当然也包括我在内，没有注释的话相信不少朋友会很晕，看得很吃力，除非公司能有非常 好的文档与足够的时间给予学习。这可能也是我这菜鸟水平，没办法进那些高大上公司的原因，呵呵.....

讲了一大堆看似无关的事情，其实主要目的是告诉初学者们，开发规范比你们想象中还重要，所以接下来就不要再懒惰了，从现在起认真要求自己，养成良好的开发习惯，对你的职业生涯的帮助将会非常大。

那么本框架开发中，我们要使用什么样的开发规范呢?

网上很多规范内容都非常详细，并不一定适合本本项目，所以针对本项目的需求我 对相关的规范文档进行了一些修改。对于本文档你只要了解就可以了，具体到编码时你按我代码中的风格尝试去编写，慢慢你就可以掌握大部分的要求了，至于更详 细更规范的要求，等本项目完成后大家再继续自己去进修吧o(∩_∩)o

目　　录

作者：AllEmpty

深入剖析 REDIS AOF 持久化策略

作者/郑思愿daoluan

Quite simple.

本篇主要讲的是 AOF 持久化，了解 AOF 的数据组织方式和运作机制。redis 主要在 aof.c 中实现 AOF 的操作。

数据结构 rio

redis AOF 持久化同样借助了 struct rio. 详细内容在《深入剖析 redis RDB 持久化策略》中有介绍。

AOF 数据组织方式

假设 redis 内存有「name:Jhon」的键值对，那么进行 AOF 持久化后，AOF 文件有如下内容：

```
*2      # 2个参数
$6      # 第一个参数长度为 6
SELECT      # 第一个参数
$1      # 第二参数长度为 1
8      # 第二参数
*3      # 3个参数
$3      # 第一个参数长度为 4
SET      # 第一个参数
$4      # 第二参数长度为 4
name      # 第二个参数
$4      # 第三个参数长度为 4
Jhon      # 第二参数长度为 4
```

所以对上面的内容进行恢复，能得到熟悉的一条 redis 命令：SELECT 8;SET name Jhon.
可以想象的是，redis 遍历内存数据集中的每个 key-value 对，依次写入磁盘中；redis 启动的时候，从 AOF 文件中读取数据，恢复数据。

AOF 持久化运作机制

和 redis RDB 持久化运作机制不同，redis AOF 有后台执行和边服务边备份两种方式。

1）AOF 后台执行的方式和 RDB 有类似的地方，fork 一个子进程，主进程仍进行服务，子进程执行 AOF 持久化，数据被 dump 到磁盘上。与 RDB 不同的是，后台子进程持久化过程中，主进程会记录期间的所有数据变更（主进程还在服务），并存储在 server.aof_rewrite_buf_blocks 中；后台子进程结束后，redis 更新缓存追加到 AOF 文件中，是 RDB 持久化所不具备的。

来说说更新缓存这个东西。redis 服务器产生数据变更的时候，譬如 set name Jhon，不仅仅会修改内存数据集，也会记

录此更新（修改）操作，记录的方式就是上面所说的数据组织方式。

更新缓存可以存储在 server.aof_buf 中，你可以把它理解为一个小型临时中转站，所有累积的更新缓存都会先放入这里，它会在特定时机写入文件或者插入到 server.aof_rewrite_buf_blocks 下链表（下面会详述）；server.aof_buf 中的数据在 propagrate() 添加，在涉及数据更新的地方都会调用 propagrate() 以累积变更。更新缓存也可以存储在 server.aof_rewrite_buf_blocks，这是一个元素类型为 struct aofrwblock 的链表，你可以把它理解为一个仓库，当后台有 AOF 子进程的时候，会将累积的更新缓存（在 server.aof_buf 中）插入到链表中，而当 AOF 子进程结束，它会被整个写入到文件。两者是有关联的。

下面是后台执行的主要代码：

```
// 启动后台子进程，执行 AOF 持久化操作。bgrewriteaofCommand()，startAppendOnly()，
serverCron() 中会调用此函数
/* This is how rewriting of the append only file in background works:
*
* 1) The user calls BGREWRITEAOF
* 2) Redis calls this function, that forks():
*    2a) the child rewrite the append only file in a temp file.
*    2b) the parent accumulates differences in server.aof_rewrite_buf.
* 3) When the child finished '2a' exists.
* 4) The parent will trap the exit code, if it's OK, will append the
*    data accumulated into server.aof_rewrite_buf into the temp file, and
*    finally will rename(2) the temp file in the actual file name.
*    The the new file is reopened as the new append only file. Profit!
*/
int rewriteAppendOnlyFileBackground(void) {
    pid_t childpid;
    long long start;

    // 已经有正在执行备份的子进程
    if (server.aof_child_pid != -1) return REDIS_ERR;

    start = ustime();
    if ((childpid = fork()) == 0) {
        char tmpfile[256];

        // 子进程
        /* Child */

        // 关闭监听
        closeListeningSockets(0);

        // 设置进程 title
        redisSetProcTitle("redis-aof-rewrite");

        // 临时文件名
        snprintf(tmpfile,256,"temp-rewriteaof-bg-%d.aof", (int) getpid());

        // 脏数据，其实就是子进程所消耗的内存大小
```

```
    if (rewriteAppendOnlyFile(tmpfile) == REDIS_OK) {
        // 获取脏数据大小
        size_t private_dirty = zmalloc_get_private_dirty();

        // 记录脏数据
        if (private_dirty) {
            redisLog(REDIS_NOTICE,
                "AOF rewrite: %zu MB of memory used by copy-on-write",
                private_dirty/(1024*1024));
        }
        exitFromChild(0);
    } else {
        exitFromChild(1);
    }
} else {
    /* Parent */
    server.stat_fork_time = ustime()-start;
    if (childpid == -1) {
        redisLog(REDIS_WARNING,
            "Can't rewrite append only file in background: fork: %s",
            strerror(errno));
        return REDIS_ERR;
    }
    redisLog(REDIS_NOTICE,
        "Background append only file rewriting started by pid %d",childpid);
        // AOF 已经开始执行，取消 AOF 计划
        server.aof_rewrite_scheduled = 0;

        // AOF 最近一次执行的起始时间
        server.aof_rewrite_time_start = time(NULL);

        // 子进程 ID
        server.aof_child_pid = childpid;
        updateDictResizePolicy();

        // 因为更新缓存都将写入文件，要强制产生选择数据集的指令 SELECT ，以防出现数据合并错
误。
        /* We set appendseldb to -1 in order to force the next call to the
         * feedAppendOnlyFile() to issue a SELECT command, so the differences
         * accumulated by the parent into server.aof_rewrite_buf will start
         * with a SELECT statement and it will be safe to merge. */
        server.aof_selected_db = -1;

        replicationScriptCacheFlush();
        return REDIS_OK;
    }
    return REDIS_OK; /* unreached */
}

// AOF 持久化主函数。只在 rewriteAppendOnlyFileBackground() 中会调用此函数
```

```
/* Write a sequence of commands able to fully rebuild the dataset into
* "filename". Used both by REWRITEAOF and BGREWRITEAOF.
*
* In order to minimize the number of commands needed in the rewritten
* log Redis uses variadic commands when possible, such as RPUSH, SADD
* and ZADD. However at max REDIS_AOF_REWRITE_ITEMS_PER_CMD items per time
* are inserted using a single command. */
int rewriteAppendOnlyFile(char *filename) {
    dictIterator *di = NULL;
    dictEntry *de;
    rio aof;
    FILE *fp;
    char tmpfile[256];
    int j;
    long long now = mstime();

    /* Note that we have to use a different temp name here compared to the
     * one used by rewriteAppendOnlyFileBackground() function. */
    snprintf(tmpfile,256,"temp-rewriteaof-%d.aof", (int) getpid());

    // 打开文件
    fp = fopen(tmpfile,"w");
    if (!fp) {
        redisLog(REDIS_WARNING, "Opening the temp file for AOF rewrite in re-
writeAppendOnlyFile(): %s", strerror(errno));
        return REDIS_ERR;
}

// 初始化 rio 结构体
rioInitWithFile(&aof,fp);

// 如果设置了自动备份参数，将进行设置
if (server.aof_rewrite_incremental_fsync)
    rioSetAutoSync(&aof,REDIS_AOF_AUTOSYNC_BYTES);

// 备份每一个数据集
for (j = 0; j < server.dbnum; j++) {
    char selectcmd[] = "*2\r\n$6\r\nSELECT\r\n";
    redisDb *db = server.db+j;
    dict *d = db->dict;
    if (dictSize(d) == 0) continue;

    // 获取数据集的迭代器
    di = dictGetSafeIterator(d);
    if (!di) {
        fclose(fp);
        return REDIS_ERR;
    }

    // 写入 AOF 操作码
```

```
/* SELECT the new DB */
if (rioWrite(&aof,selectcmd,sizeof(selectcmd)-1) == 0) goto werr;

// 写入数据集序号
if (rioWriteBulkLongLong(&aof,j) == 0) goto werr;

// 写入数据集中每一个数据项
/* Iterate this DB writing every entry */
while((de = dictNext(di)) != NULL) {
    sds keystr;
    robj key, *o;
    long long expiretime;

    keystr = dictGetKey(de);
    o = dictGetVal(de);

    // 将 keystr 封装在 robj 里
    initStaticStringObject(key,keystr);

    // 获取过期时间
    expiretime = getExpire(db,&key);

    // 如果已经过期，放弃存储
    /* If this key is already expired skip it */
    if (expiretime != -1 && expiretime < now) continue;

// 写入键值对应的写操作
/* Save the key and associated value */
if (o->type == REDIS_STRING) {
    /* Emit a SET command */
    char cmd[]="*3\r\n$3\r\nSET\r\n";
    if (rioWrite(&aof,cmd,sizeof(cmd)-1) == 0) goto werr;
    /* Key and value */
    if (rioWriteBulkObject(&aof,&key) == 0) goto werr;
    if (rioWriteBulkObject(&aof,o) == 0) goto werr;
} else if (o->type == REDIS_LIST) {
    if (rewriteListObject(&aof,&key,o) == 0) goto werr;
} else if (o->type == REDIS_SET) {
    if (rewriteSetObject(&aof,&key,o) == 0) goto werr;
} else if (o->type == REDIS_ZSET) {
    if (rewriteSortedSetObject(&aof,&key,o) == 0) goto werr;
} else if (o->type == REDIS_HASH) {
    if (rewriteHashObject(&aof,&key,o) == 0) goto werr;
} else {
    redisPanic("Unknown object type");
}

// 写入过期时间
/* Save the expire time */
if (expiretime != -1) {
```

```
                char cmd[]="*3\r\n$9\r\nPEXPIREAT\r\n";
                if (rioWrite(&aof,cmd,sizeof(cmd)-1) == 0) goto werr;
                if (rioWriteBulkObject(&aof,&key) == 0) goto werr;
                if (rioWriteBulkLongLong(&aof,expiretime) == 0) goto werr;
            }
        }

        // 释放迭代器
        dictReleaseIterator(di);
    }

    // 写入磁盘
    /* Make sure data will not remain on the OS's output buffers */
    fflush(fp);
    aof_fsync(fileno(fp));
    fclose(fp);

    // 重写文件名
    /* Use RENAME to make sure the DB file is changed atomically only
     * if the generate DB file is ok. */
    if (rename(tmpfile,filename) == -1) {
        redisLog(REDIS_WARNING,"Error moving temp append only file on the final
destination: %s", strerror(errno));
        unlink(tmpfile);
        return REDIS_ERR;
    }
    redisLog(REDIS_NOTICE,"SYNC append only file rewrite performed");
    return REDIS_OK;

werr:
    // 清理工作
    fclose(fp);
    unlink(tmpfile);
    redisLog(REDIS_WARNING,"Write error writing append only file on disk: %s",
strerror(errno));
    if (di) dictReleaseIterator(di);
    return REDIS_ERR;
}

// 后台子进程结束后，redis 更新缓存 server.aof_rewrite_buf_blocks 追加到 AOF 文件中
// 在 AOF 持久化结束后会执行这个函数， backgroundRewriteDoneHandler() 主要工作是将
server.aof_rewrite_buf_blocks，即 AOF 缓存写入文件
/* A background append only file rewriting (BGREWRITEAOF) terminated its work.
 * Handle this. */
void backgroundRewriteDoneHandler(int exitcode, int bysignal) {
        ......
        // 将 AOF 缓存 server.aof_rewrite_buf_blocks 的 AOF 写入磁盘
        if (aofRewriteBufferWrite(newfd) == -1) {
            redisLog(REDIS_WARNING,
```

```
                "Error trying to flush the parent diff to the rewritten AOF:
%s", strerror(errno));
            close(newfd);
            goto cleanup;
        }
        ......
}

// 将累积的更新缓存 server.aof_rewrite_buf_blocks 同步到磁盘
/* Write the buffer (possibly composed of multiple blocks) into the specified
* fd. If no short write or any other error happens -1 is returned,
* otherwise the number of bytes written is returned. */
ssize_t aofRewriteBufferWrite(int fd) {
    listNode *ln;
    listIter li;
    ssize_t count = 0;

    listRewind(server.aof_rewrite_buf_blocks,&li);
    while((ln = listNext(&li))) {
        aofrwblock *block = listNodeValue(ln);
        ssize_t nwritten;

        if (block->used) {
            nwritten = write(fd,block->buf,block->used);
            if (nwritten != block->used) {
                if (nwritten == 0) errno = EIO;
                return -1;
            }
            count += nwritten;
        }
    }
    return count;
}
```

2）边服务边备份的方式，即 redis 服务器会把所有的数据变更存储在 server.aof_buf 中，并在特定时机将更新缓存写入预设定的文件（server.aof_filename）。特定时机有三种：

1. 进入事件循环之前
2. redis 服务器定时程序 serverCron() 中
3. 停止 AOF 策略的 stopAppendOnly() 中

redis 无非是不想服务器突然崩溃终止，导致过多的数据丢失。redis 默认是每两秒钟进行一次边服务边备份，即隔两秒将累积的写入文件。

redis 为什么取消直接在本进程进行 AOF 持久化的方法？原因可能是产生一个 AOF 文件要比 RDB 文件消耗更多的时间；如果在当前进程执行 AOF 持久化，会占用服务进程（主进程）较多的时间，停止服务的时间也更长（？）

下面是边服务边备份的主要代码：

```
// 同步磁盘；将所有累积的更新 server.aof_buf 写入磁盘
/* Write the append only file buffer on disk.
```

```
*
* Since we are required to write the AOF before replying to the client,
* and the only way the client socket can get a write is entering when the
* the event loop, we accumulate all the AOF writes in a memory
* buffer and write it on disk using this function just before entering
* the event loop again.
*
* About the 'force' argument:
*
* When the fsync policy is set to 'everysec' we may delay the flush if there
* is still an fsync() going on in the background thread, since for instance
* on Linux write(2) will be blocked by the background fsync anyway.
* When this happens we remember that there is some aof buffer to be
* flushed ASAP, and will try to do that in the serverCron() function.
*
* However if force is set to 1 we'll write regardless of the background
* fsync. */
void flushAppendOnlyFile(int force) {
    ssize_t nwritten;
    int sync_in_progress = 0;

    // 无数据，无需同步到磁盘
    if (sdslen(server.aof_buf) == 0) return;

    // 创建线程任务，主要调用 fsync()
    if (server.aof_fsync == AOF_FSYNC_EVERYSEC)
        sync_in_progress = bioPendingJobsOfType(REDIS_BIO_AOF_FSYNC) != 0;

    // 如果没有设置强制同步的选项，可能不会立即进行同步
    if (server.aof_fsync == AOF_FSYNC_EVERYSEC && !force) {
        // 推迟执行 AOF
        /* With this append fsync policy we do background fsyncing.
         * If the fsync is still in progress we can try to delay
         * the write for a couple of seconds. */
        if (sync_in_progress) {
            if (server.aof_flush_postponed_start == 0) {
                // 设置延迟冲洗时间选项
                /* No previous write postponinig, remember that we are
                 * postponing the flush and return. */
                server.aof_flush_postponed_start = server.unixtime; // /* Unix
time sampled every cron cycle. */
                return;

            // 没有超过 2s，直接结束
            } else if (server.unixtime - server.aof_flush_postponed_start < 2) {
                /* We were already waiting for fsync to finish, but for less
                 * than two seconds this is still ok. Postpone again. */
                return;
            }
```

```
            // 否则，要强制写入磁盘
            /* Otherwise fall trough, and go write since we can't wait
             * over two seconds. */
            server.aof_delayed_fsync++;
            redisLog(REDIS_NOTICE,"Asynchronous AOF fsync is taking too long
(disk is busy?). Writing the AOF buffer without waiting for fsync to complete,
this may slow down Redis.");
        }
    }

    // 取消延迟冲洗时间设置
    /* If you are following this code path, then we are going to write so
     * set reset the postponed flush sentinel to zero. */
    server.aof_flush_postponed_start = 0;

    /* We want to perform a single write. This should be guaranteed atomic
     * at least if the filesystem we are writing is a real physical one.
     * While this will save us against the server being killed I don't think
     * there is much to do about the whole server stopping for power problems
     * or alike */
    // AOF 文件已经打开了。将 server.aof_buf 中的所有缓存数据写入文件
    nwritten = write(server.aof_fd,server.aof_buf,sdslen(server.aof_buf));

    if (nwritten != (signed)sdslen(server.aof_buf)) {
        /* Ooops, we are in troubles. The best thing to do for now is
         * aborting instead of giving the illusion that everything is
         * working as expected. */
        if (nwritten == -1) {
            redisLog(REDIS_WARNING,"Exiting on error writing to the append-only
file: %s",strerror(errno));
        } else {
            redisLog(REDIS_WARNING,"Exiting on short write while writing to "
                                   "the append-only file: %s (nwritten=%ld, "
                                   "expected=%ld)",
                                   strerror(errno),
                                   (long)nwritten,
                                   (long)sdslen(server.aof_buf));

            if (ftruncate(server.aof_fd, server.aof_current_size) == -1) {
                redisLog(REDIS_WARNING, "Could not remove short write "
                         "from the append-only file.  Redis may refuse "
                         "to load the AOF the next time it starts.  "
                         "ftruncate: %s", strerror(errno));
            }
        }
        exit(1);
    }

    // 更新 AOF 文件的大小
    server.aof_current_size += nwritten;
```

```
    /*当 server.aof_buf 足够小,重新利用空间，防止频繁的内存分配。
    相反，当 server.aof_buf 占据大量的空间，采取的策略是释放空间，可见 redis 对内存很敏感。
*/
    /* Re-use AOF buffer when it is small enough. The maximum comes from the
     * arena size of 4k minus some overhead (but is otherwise arbitrary). */
    if ((sdslen(server.aof_buf)+sdsavail(server.aof_buf)) < 4000) {
        sdsclear(server.aof_buf);
    } else {
        sdsfree(server.aof_buf);
        server.aof_buf = sdsempty();
    }

    /* Don't fsync if no-appendfsync-on-rewrite is set to yes and there are
     * children doing I/O in the background. */
    if (server.aof_no_fsync_on_rewrite &&
        (server.aof_child_pid != -1 || server.rdb_child_pid != -1))
            return;

    // sync,写入磁盘
    /* Perform the fsync if needed. */
    if (server.aof_fsync == AOF_FSYNC_ALWAYS) {
        /* aof_fsync is defined as fdatasync() for Linux in order to avoid
         * flushing metadata. */
        aof_fsync(server.aof_fd); /* Let's try to get this data on the disk */
        server.aof_last_fsync = server.unixtime;
    } else if ((server.aof_fsync == AOF_FSYNC_EVERYSEC &&
                server.unixtime > server.aof_last_fsync)) {
        if (!sync_in_progress) aof_background_fsync(server.aof_fd);
        server.aof_last_fsync = server.unixtime;
    }
}
```

细说更新缓存

上面两次提到了「更新缓存」，它即是 redis 累积的数据变更。

更新缓存可以存储在 server.aof_buf 中，可以存储在 server.server.aof_rewrite_buf_blocks 连表中。他们的关系是：每一次数据变更记录都会写入 server.aof_buf 中，同时如果后台子进程在持久化，变更记录还会被写入 server.server.aof_rewrite_buf_blocks 中。server.aof_buf 会在特定时期写入指定文件，server.server.aof_rewrite_buf_blocks 会在后台持久化结束后追加到文件。

redis 源码中是这么实现的：propagrate()->feedAppendOnlyFile()->aofRewriteBufferAppend()

注释：feedAppendOnlyFile() 会把更新添加到 server.aof_buf；接下来会有一个判断，如果存在 AOF 子进程，则调用 aofRewriteBufferAppend() 将 server.aof_buf 中的所有数据插入到 server.aof_rewrite_buf_blocks 链表。

一副可以缓解视力疲劳的图片——AOF 持久化运作机制：

下面是主要的代码:

```
// 向 AOF 和从机发布数据更新
/* Propagate the specified command (in the context of the specified database id)
* to AOF and Slaves.
*
* flags are an xor between:
* + REDIS_PROPAGATE_NONE (no propagation of command at all)
* + REDIS_PROPAGATE_AOF (propagate into the AOF file if is enabled)
* + REDIS_PROPAGATE_REPL (propagate into the replication link)
*/
void propagate(struct redisCommand *cmd, int dbid, robj **argv, int argc,
               int flags)
{
    // AOF 策略需要打开，且设置 AOF 传播标记，将更新发布给本地文件
    if (server.aof_state != REDIS_AOF_OFF && flags & REDIS_PROPAGATE_AOF)
        feedAppendOnlyFile(cmd,dbid,argv,argc);

    // 设置了从机传播标记，将更新发布给从机
    if (flags & REDIS_PROPAGATE_REPL)
        replicationFeedSlaves(server.slaves,dbid,argv,argc);
}

// 将数据更新记录到 AOF 缓存中
void feedAppendOnlyFile(struct redisCommand *cmd, int dictid, robj **argv, int
argc) {
    sds buf = sdsempty();
    robj *tmpargv[3];

    /* The DB this command was targeting is not the same as the last command
     * we appendend. To issue a SELECT command is needed. */
    if (dictid != server.aof_selected_db) {
        char seldb[64];

        snprintf(seldb,sizeof(seldb),"%d",dictid);
        buf = sdscatprintf(buf,"*2\r\n$6\r\nSELECT\r\n$%lu\r\n%s\r\n",
            (unsigned long)strlen(seldb),seldb);
        server.aof_selected_db = dictid;
    }

    if (cmd->proc == expireCommand || cmd->proc == pexpireCommand ||
        cmd->proc == expireatCommand) {
        /* Translate EXPIRE/PEXPIRE/EXPIREAT into PEXPIREAT */
        buf = catAppendOnlyExpireAtCommand(buf,cmd,argv[1],argv[2]);
    } else if (cmd->proc == setexCommand || cmd->proc == psetexCommand) {
        /* Translate SETEX/PSETEX to SET and PEXPIREAT */
        tmpargv[0] = createStringObject("SET",3);
        tmpargv[1] = argv[1];
        tmpargv[2] = argv[3];
        buf = catAppendOnlyGenericCommand(buf,3,tmpargv);
```

```
        decrRefCount(tmpargv[0]);
        buf = catAppendOnlyExpireAtCommand(buf,cmd,argv[1],argv[2]);
    } else {
        /* All the other commands don't need translation or need the
         * same translation already operated in the command vector
         * for the replication itself. */
        buf = catAppendOnlyGenericCommand(buf,argc,argv);
    }

    // 将生成的 AOF 追加到 server.aof_buf 中。server.在下一次进入事件循环之前，aof_buf
中的内容将会写到磁盘上
    /* Append to the AOF buffer. This will be flushed on disk just before
     * of re-entering the event loop, so before the client will get a
     * positive reply about the operation performed. */
    if (server.aof_state == REDIS_AOF_ON)
        server.aof_buf = sdscatlen(server.aof_buf,buf,sdslen(buf));

    // 如果已经有 AOF 子进程运行，redis 采取的策略是累积子进程 AOF 备份的数据和内存中数据集
的差异。 aofRewriteBufferAppend() 把 buf 的内容追加到 server.aof_rewrite_buf_blocks
数组中
    /* If a background append only file rewriting is in progress we want to
     * accumulate the differences between the child DB and the current one
     * in a buffer, so that when the child process will do its work we
     * can append the differences to the new append only file. */
    if (server.aof_child_pid != -1)
        aofRewriteBufferAppend((unsigned char*)buf,sdslen(buf));

    sdsfree(buf);
}

// 将数据更新记录写入 server.aof_rewrite_buf_blocks，此函数只由 feedAppendOnlyFile()
调用
/* Append data to the AOF rewrite buffer, allocating new blocks if needed. */
void aofRewriteBufferAppend(unsigned char *s, unsigned long len) {
    // 尾插法
    listNode *ln = listLast(server.aof_rewrite_buf_blocks);
    aofrwblock *block = ln ? ln->value : NULL;

    while(len) {
        /* If we already got at least an allocated block, try appending
         * at least some piece into it. */
        if (block) {
            unsigned long thislen = (block->free < len) ? block->free : len;
            if (thislen) {  /* The current block is not already full. */
                memcpy(block->buf+block->used, s, thislen);
                block->used += thislen;
                block->free -= thislen;
                s += thislen;
                len -= thislen;
            }
```

```
        }

        if (len) { /* First block to allocate, or need another block. */
            int numblocks;

            // 创建新的节点，插到尾部
            block = zmalloc(sizeof(*block));
            block->free = AOF_RW_BUF_BLOCK_SIZE;
            block->used = 0;

            // 尾插法
            listAddNodeTail(server.aof_rewrite_buf_blocks,block);

            /* Log every time we cross more 10 or 100 blocks, respectively
             * as a notice or warning. */
            numblocks = listLength(server.aof_rewrite_buf_blocks);
            if (((numblocks+1) % 10) == 0) {
                int level = ((numblocks+1) % 100) == 0 ? REDIS_WARNING :
                                                         REDIS_NOTICE;
                redisLog(level,"Background AOF buffer size: %lu MB",
                    aofRewriteBufferSize()/(1024*1024));
            }
        }
    }
}
```

两种数据落地的方式，就是 AOF 的两个主线。因此，redis AOF 持久化机制有两条主线：后台执行和边服务边备份，抓住这两点就能理解 redis AOF 了。

这里有一个疑问，两条主线都会涉及文件的写：后台执行会写一个 AOF 文件，边服务边备份也会写一个，以哪个为准?

后台持久化的数据首先会被写入「temp-rewriteaof-bg-%d.aof」，其中「%d」是 AOF 子进程 id；待 AOF 子进程结束后，「temp-rewriteaof-bg-%d.aof」会被以追加的方式打开，继而写入 server.aof_rewrite_buf_blocks 中的更新缓存，最后「temp-rewriteaof-bg-%d.aof」文件被命名为 server.aof_filename，所以之前的名为 server.aof_filename 的文件会被删除，也就是说边服务边备份写入的文件会被删除。边服务边备份的数据会被一直写入到 server.aof_filename 文件中。

因此，确实会产生两个文件，但是最后都会变成 server.aof_filename 文件。

这里还有一个疑问，既然有了后台持久化，为什么还要边服务边备份? 边服务边备份时间长了会产生数据冗余甚至备份过旧的数据，而后台持久化可以消除这些东西。看，这里是 redis 的双保险。

AOF 恢复过程

AOF 的数据恢复过程设计实在是棒极了，它模拟一个服务过程。redis 首先虚拟一个客户端，读取 AOF 文件恢复 redis 命令和参数；然后就像服务客户端一样执行命令相应的函数，从而恢复数据。这些过程主要在loadAppendOnlyFile() 中实现。

```
// 加载 AOF 文件，恢复数据
```

```
/* Replay the append log file. On error REDIS_OK is returned. On non fatal
* error (the append only file is zero-length) REDIS_ERR is returned. On
* fatal error an error message is logged and the program exists. */
int loadAppendOnlyFile(char *filename) {
    struct redisClient *fakeClient;
    FILE *fp = fopen(filename,"r");
    struct redis_stat sb;
    int old_aof_state = server.aof_state;
    long loops = 0;

    // 文件大小不能为 0
    if (fp && redis_fstat(fileno(fp),&sb) != -1 && sb.st_size == 0) {
        server.aof_current_size = 0;
        fclose(fp);
        return REDIS_ERR;
    }

    if (fp == NULL) {
        redisLog(REDIS_WARNING,"Fatal error: can't open the append log file for
reading: %s",strerror(errno));
        exit(1);
    }

    // 正在执行 AOF 加载操作，于是暂时禁止 AOF 的所有操作，以免混淆
    /* Temporarily disable AOF, to prevent EXEC from feeding a MULTI
     * to the same file we're about to read. */
    server.aof_state = REDIS_AOF_OFF;

    // 虚拟出一个客户端，即 redisClient
    fakeClient = createFakeClient();
    startLoading(fp);

    while(1) {
        int argc, j;
        unsigned long len;
        robj **argv;
        char buf[128];
        sds argsds;
        struct redisCommand *cmd;

        // 每循环 1000 次，在恢复数据的同时，服务器也为客户端服务。aeProcessEvents() 会进
入事件循环
        /* Serve the clients from time to time */
        if (!(loops++ % 1000)) {
            loadingProgress(ftello(fp));
            aeProcessEvents(server.el, AE_FILE_EVENTS|AE_DONT_WAIT);
        }

        // 可能 aof 文件到了结尾
        if (fgets(buf,sizeof(buf),fp) == NULL) {
```

```
        if (feof(fp))
            break;
        else
            goto readerr;
    }

    // 必须以“*”开头，格式不对，退出
    if (buf[0] != '*') goto fmterr;

    // 参数的个数
    argc = atoi(buf+1);

    // 参数个数错误
    if (argc < 1) goto fmterr;

    // 为参数分配空间
    argv = zmalloc(sizeof(robj*)*argc);

    // 依次读取参数
    for (j = 0; j < argc; j++) {
        if (fgets(buf,sizeof(buf),fp) == NULL) goto readerr;
        if (buf[0] != '$') goto fmterr;
        len = strtol(buf+1,NULL,10);
        argsds = sdsnewlen(NULL,len);
        if (len && fread(argsds,len,1,fp) == 0) goto fmterr;
        argv[j] = createObject(REDIS_STRING,argsds);
        if (fread(buf,2,1,fp) == 0) goto fmterr; /* discard CRLF */
    }

    // 找到相应的命令
    /* Command lookup */
    cmd = lookupCommand(argv[0]->ptr);
    if (!cmd) {
        redisLog(REDIS_WARNING,"Unknown command '%s' reading the append only
file", (char*)argv[0]->ptr);
        exit(1);
    }

    // 执行命令，模拟服务客户端请求的过程，从而写入数据
    /* Run the command in the context of a fake client */
    fakeClient->argc = argc;
    fakeClient->argv = argv;
    cmd->proc(fakeClient);

    /* The fake client should not have a reply */
    redisAssert(fakeClient->bufpos == 0 && listLength(fakeClient->reply) ==
0);
    /* The fake client should never get blocked */
    redisAssert((fakeClient->flags & REDIS_BLOCKED) == 0);
```

```
    // 释放虚拟客户端空间
    /* Clean up. Command code may have changed argv/argc so we use the
     * argv/argc of the client instead of the local variables. */
    for (j = 0; j < fakeClient->argc; j++)
        decrRefCount(fakeClient->argv[j]);
    zfree(fakeClient->argv);
}

/* This point can only be reached when EOF is reached without errors.
 * If the client is in the middle of a MULTI/EXEC, log error and quit. */
if (fakeClient->flags & REDIS_MULTI) goto readerr;

// 清理工作
fclose(fp);
freeFakeClient(fakeClient);

// 恢复旧的 AOF 状态
server.aof_state = old_aof_state;
stopLoading();

// 记录最近 AOF 操作的文件大小
aofUpdateCurrentSize();
server.aof_rewrite_base_size = server.aof_current_size;
return REDIS_OK;

readerr:
    // 错误，清理工作
    if (feof(fp)) {
        redisLog(REDIS_WARNING,"Unexpected end of file reading the append only
file");
    } else {
        redisLog(REDIS_WARNING,"Unrecoverable error reading the append only
file: %s", strerror(errno));
    }
    exit(1);
fmterr:
    redisLog(REDIS_WARNING,"Bad file format reading the append only file: make a
backup of your AOF file, then use ./redis-check-aof --fix <filename>");
    exit(1);
}
```

AOF 的适用场景

如果对数据比较关心，分秒必争，可以用 AOF 持久化，而且 AOF 文件很容易进行分析。

作者：@郑思愿daoluan

本文转载自:http://daoluan.net/blog/decode-redis-aof-persistence/

如何在REDIS里按模式删除数据

作者/火丁笔记
huoding.com

一台Redis服务器在很短的时间里消耗了几十个G的内存，最终因为SWAP而宕机。因为这台服务器的社会背景比较复杂，所以一时无法判断犯罪嫌疑人到底是谁。

最开始的直觉是认为肯定有人保存了大体积的数据，于是问题就变成了找出哪些键占用的空间比较大，DBA同事用了redis-rdb-tools等工具来分析数据文件。可惜的是虽然找到了一些大体积的键，但最终都排除了嫌疑，问题似乎陷入了僵局。

在被直觉带入死胡同之后，我们开始调整调查的角度：即便一个键本身占用的空间并不大，但是如果相同模式的键数量很多的话，那么合计起来一样会占用大 量空间，于是问题就变成了找出哪些相同模式的键占用的空间比较大。这次我不想用什么工具，而是打算在测试服务器上一边删除可疑键一边查看内存变化情况：

```
shell> /path/to/redis-cli keys foo:* | xargs /path/to/redis-cli del
```

悲催的是一运行这个命令服务器就挂了！因为数据太多了，所以KEYS受不了。此时应该使用SCAN，它有游标的概念，每次迭代只涉及很少的数据。

直接在命令行使用SCAN有些麻烦，于是我用了PHP：

```
<?php

$redis = new Redis();
$redis->setOption(Redis::OPT_SCAN,
Redis::SCAN_RETRY);

$match = 'foo:*';
$count = 10000;

while ($keys = $redis->scan($it, $match, $count)) {
    $redis->del($keys);
}

?>
```

在删除的同时注意监控内存变化情况，就能确认问题了：

```
shell> watch -d -n 1 '/path/to/redis-cli info | grep
memory'
```

至于可疑键的获取，我是瞎蒙的，简单通过MONITOR或者SCAN获取采样数据即可，另外从此案例看，监控键总数的变化幅度是很重要的，从INFO里能拿到它。

作者：@火丁笔记

本文转载自：http://huoding.com/2014/04/11/343

网易的SPARK技术实践

网易的实时计算需求

对于大多数的大数据而言，实时性是其所应具备的重要属性，信息的到达和获取应满足实时性的要求，而信息的价值需在其到达那刻展现才能利益最大化，例如电商网站，网站推荐系统期望能实时根据顾客的点击行为分析其购买意愿，做到精准营销。

实时计算指针对只读（Read Only）数据进行即时数据的获取和计算，也可以成为在线计算，在线计算的实时级别分为三类：Real-Time(msec/sec级)、Near Real-Time(min/hours)以及Batch(days)。 在批处理方面，MapReduce(MR)已经证明其为最有效的工具，随着MR的开源实现Hadoop为代表的大数据分析技术的普及，其在大处理方面的能 力已经得到认可，但是它更适用于对集群上大数据的批处理，并不适用于实时处理大规模流数据。为了满足实时性的要求，基于数据仓库所构建的流计算和实时性计 算框架也不断涌现，相关围绕MR的实时性优化技术也蓬勃发展，比较代表性的系统Google Dremel、Twitter Storm以及Yahoo S4等。

大数据的应用类型主要分为：批处理（Batch Processing）和流处理（Stream Processing）两方面。批处理是先存储后处理（Store-Then-Process）,流处理是直接处理（Straight-Through- Processing）,为提高商业智能的反映时间，目前广泛所采取的大数据处理框架,例如MR和Dryad所面向的主要是大规模数据分析，以批处理计算 为主，其实时性需求得不到满足。常用的应用有在线推荐、网页点击分析、传感网络、交通分析以及金融中的高频交易，对实时分析处理（Real Time Analytic Processing, RTAP）的需求越来显著，网易公司作为国内最大的门户网站之一，实时性也是公司目前互联网产品所应具备的重要属性。

网易大数据Spark技术应用

Spark技术代表未来数据处理的新方向，Spark是UC Berkeley AMP lab开源的类Hadoop MapReduce的通用并行计算框架，Spark基于MapReduce实现分布式计算，拥有Hadoop MapReduce具有的优点。不同于MapReduce的是，Job中间

输出和结果可以保存在内存中，从而不再需要读写HDFS，因此Spark能更好 地适用于数据挖掘与机器学习等需要迭代的MapReduce的算法。

在网易大数据平台中，数据存储在HDFS之后，提供Hive的数据仓库计算和查询，要提高数据处理的性能并达到实时级别，网易公司采用的是 Impala和Shark结合的混合实时技术。Cloudera Impala是基于Hadoop的实时检索引擎开源项目，其效率比Hive提高3-90倍，其本质是Google Dremel的模仿，但在SQL功能上青出于蓝胜于蓝。Shark是基于Spark的SQL实现，Shark可以比Hive 快40倍（其论文所描述）， 如果执行机器学习程序，可以快 25倍，并完全和Hive兼容。

图1和图2分别测试的计算能力和实时查询性能经过初步测试，在网易的实时计算平台，在大数据实时查询系统中，Impala在数据处理方面的速度可以 相比HIVE达到3倍到30倍的加速比，Shark可以相比HIVE达到 1.5到15倍的加速比，相比较Impala和Shark引擎，通常Impala会比Shark快一倍，这里可能会引出思考，既然Impala实时性如此 好，为何还需要Shark呢?

在设计大数据平台的时候，我们发现Impala性能不错，但是其对旧Hive的数据不兼容，因为目前的大数据应用中很多都是Hive的组织方式，而 Shark可以完全兼容旧的数据，因此在目前的数据结构中必须采用混合的数据处理模式。Hive和Impala会协同存在一段时间， Hive主要为Predefined Queries，并主要处理批处理相关作业，而Impala则处理交互的查询（AD-HOC Queries），使得大数据系统既支持OLTP，也支持OLAP,以达到实时分析处理（Real Time Analytic Processing, RTAP）的水平。

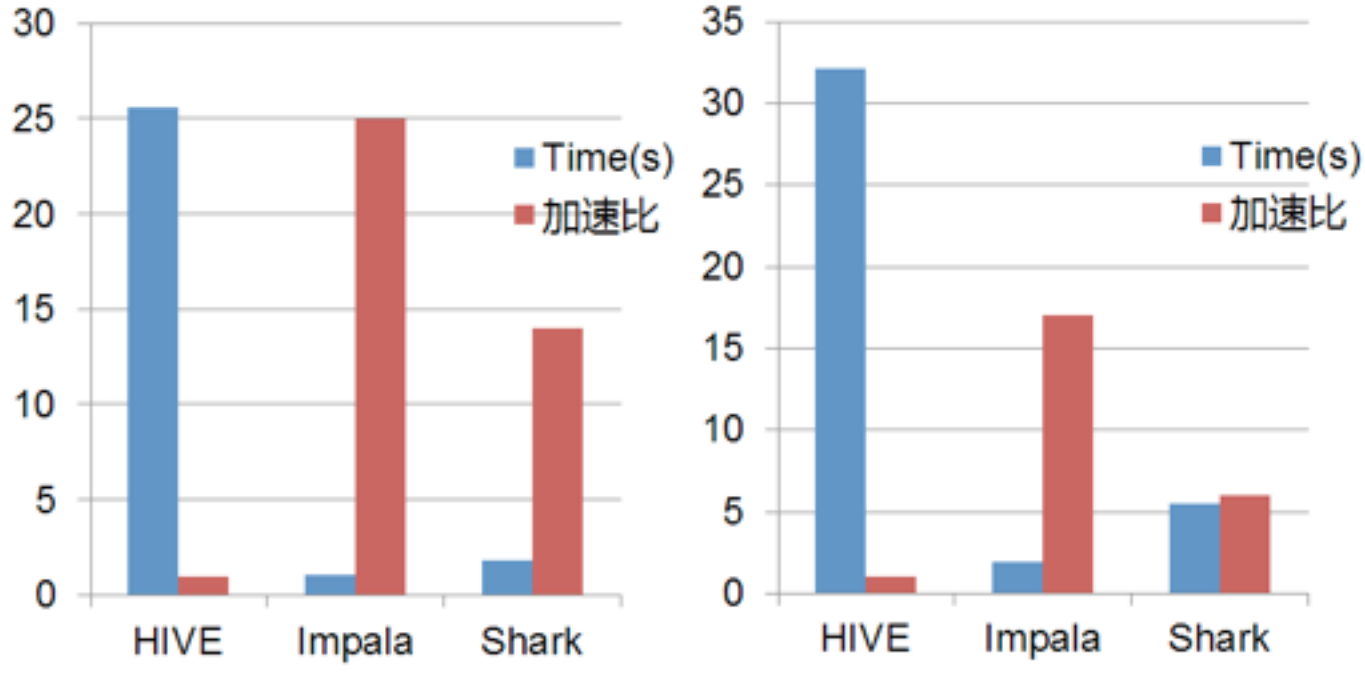


图1 网易大数据平台性能测试(Count/Sum/Avg操作)

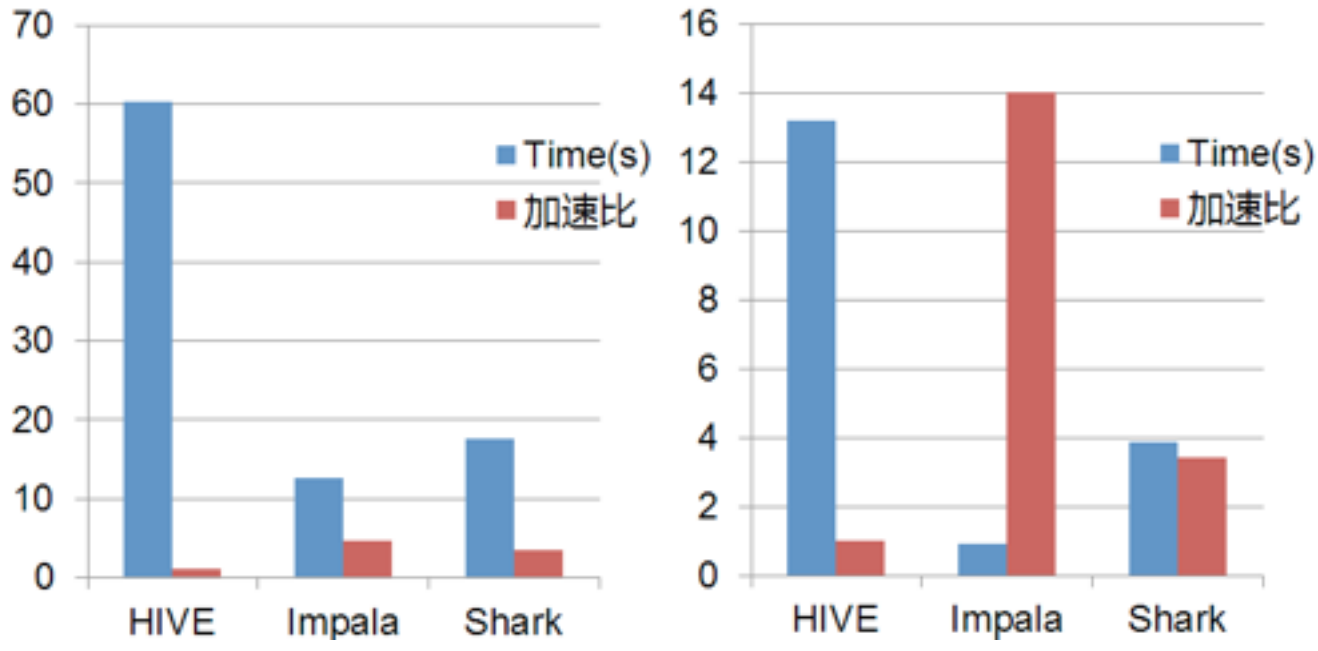


图2网易大数据平台性能测试(Join/Ad-hoc查询操作)

总结

如果要评价2012到2013年度IT业界热词，非“大数据”一词莫属。ROI（Return On Investment）投资回报率已经演化为Return On Information,信息的回报率成为互联网公司的一个重要指标，如果所掌握的海量数据都是一堆“垃圾”，没有金矿去挖掘，那大数据也无从谈起，而提 高ROI的一个重要属性就是实时性，提高数据的反应时间需要技术做支撑和保障，网易作为中国顶尖的互联网公司之一，在大数据方面也是最早的先行者，特别实 时计算技术方面，公司很早就开始采用最新的技术来提供服务，例如Impala和Shark等，不难发现，网易的大数据系统可以灵活的选择计算实时引擎，总 体上系统在实时处理方面的能力可以提升2到15倍，这对于提升公司的生产效率有显著成　效，在后续的工作中期望可以进一步提升实时级别，目前只能做到秒级， 能否达到毫秒级甚至微秒级别是将来的一个研发方向,总之对于海量数据计算、实时性方面有强烈需求的公司应用落地Spark是很好的选择。

参考资料

[1] Storm Distributed and fault-tolerant real time computation

[2] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari. S4: Distributed Stream Computing Platform. 2010 IEEE International Conference on Data Mining Workshops (ICDMW).

[3] Cloudera Impala https://github.com/cloudera/impala

Reynold S. Xin, Josh Rosen, et al. Shark: SQL and rich analytics at scale. SIGMOD Conference 2013.

作者：王健宗

本文转载自：http://www.infoq.com/cn/news/2014/04/netease-spark-practice

文章： 豆瓣的基础架构

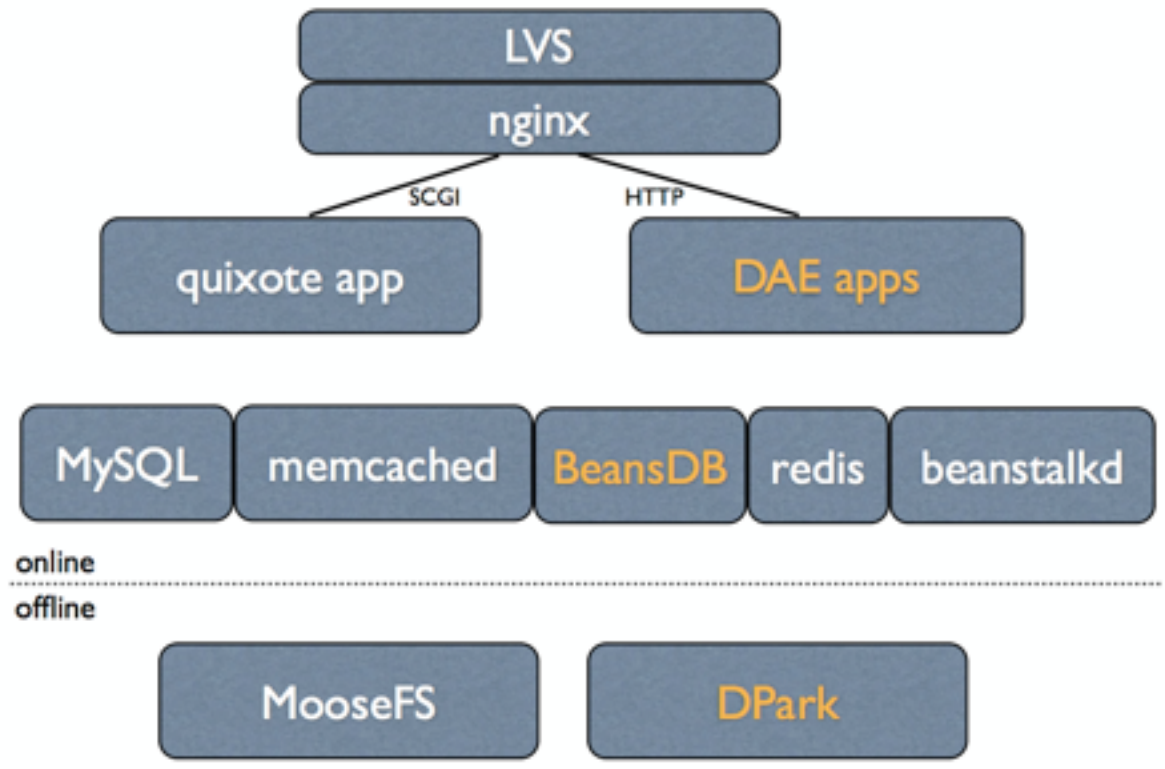
本文根据InfoQ中文站对豆瓣洪强宁（@hongqn）的沟通交流整理而成。洪强宁介绍了豆瓣的架构和组件，并分享了豆瓣基础平台部的一些团队经验。文中截图来自洪强宁在2013年CTO俱乐部中的分享。

作者/hongqn
Python Programmer

嘉宾介绍

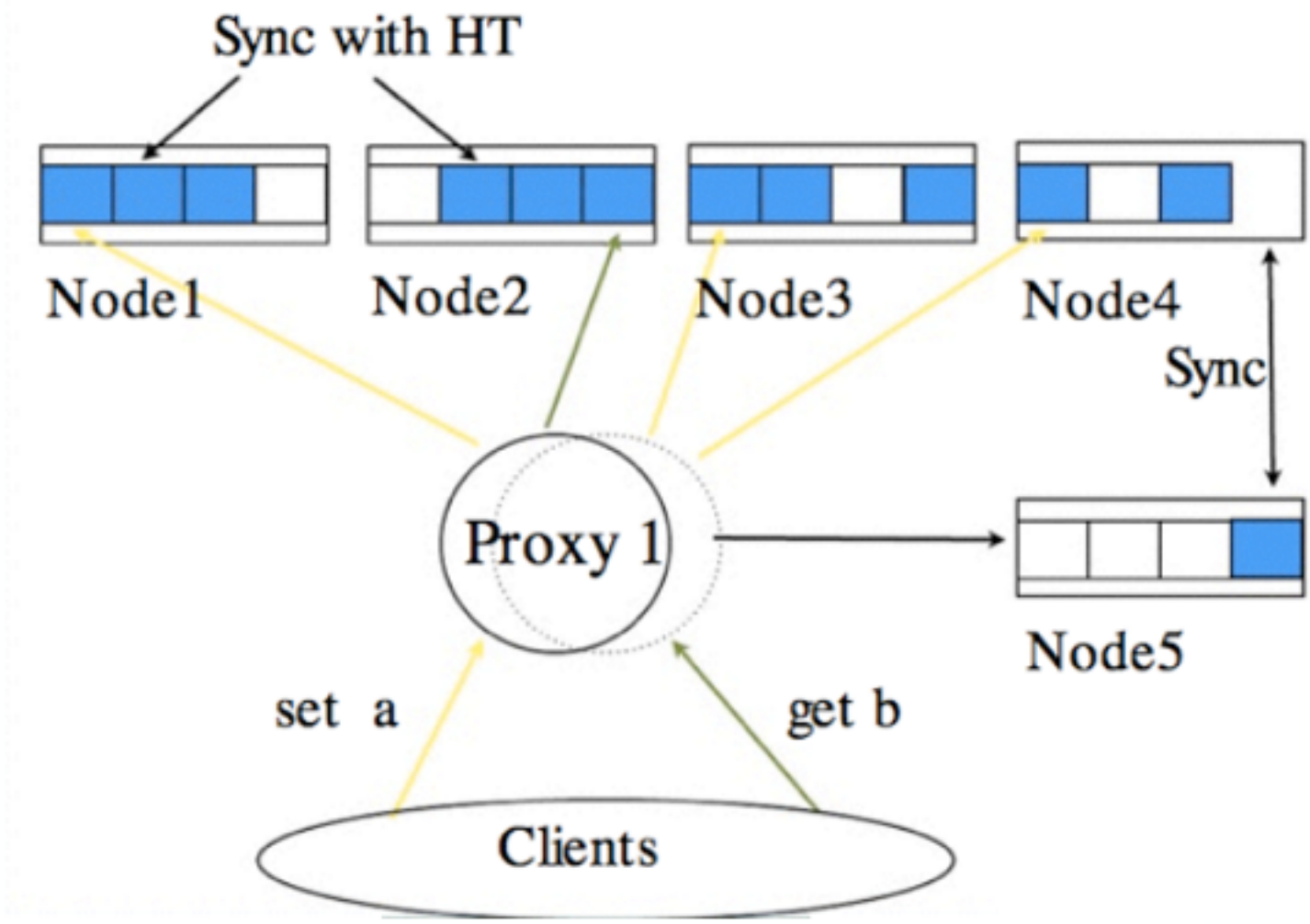
洪强宁，豆瓣首席架构师。豆瓣第一位全职员工。清华毕业后，洪强宁一直做嵌入式系统。在2002年开始接触Python语言，从硬件工程师变为软件工程师，对一种语言在计算机底层如何工作有深入的理解。

架构



豆瓣整个基础架构可以粗略的分为在线和离线两大块。在线的部分和大部分网站类似：前面用LVS做HA，用Nginx做反向代理，形成负载均衡的一 层；应用层主要是做运算，将运算结果返回给前面的用户，DAE平台是这两年建起来的，现在大部分豆瓣的应用基本都跑在DAE上面了；应用后面的基础服务也 跟其他网站差不多，MySQL、memcached、redis、beanstalkd，不一样的是NoSQL的选择——BeansDB，这是我们在几年前开源的KV数据库，也是国内比较早开源的KV数据库。

BeansDB项目可以说是一个简化版的AWS DynamoD-
B，该项目在2008年启动，2009年开源，第一版使用tokyo cabinet作为存储引擎，2010年使用bitcask存储格式重写了存储引擎，性能更好。BeansDB对key做哈希运算找到节点来实现分布和冗 余， 一个写操作会写好几个节点，而现在的配置是写三份读一份。BeansDB主要的特点是支持海量KV数据库——相比Redis这种支持几十个G到几百个G的 内存KV数据库，BeansDB可以支持到上百T的数据。另外BeansDB最大的好处就是运维很简单，性能、可用性、扩容都很好，也实现了最终一致性。



BeansDB中间的Proxy是用Go语言写的，也是一个开源的组件。整体来说BeansDB的设计结构比较简单，相比Redis那种有多种value 类型的方式，BeansDB的Value比较简单一些。

在豆瓣内部建立了两个不同的BeansDB集群，一个是doubandb，一个是doubanfs，分别针对不同的场景。doubandb主要存储 小型文本数据，如影评、用户个人介绍、帖子内容等，这样的好处是可以大大降低我们对MySQL的性能依赖，算是给MySQL减负；doubanfs主要存 放图片和音频等中型数据。

DAE可以说是基于很多以前积累的、旧的组件做起来的。我们做的这种对内的PaaS，相比对外的PaaS而言做了很多简化，尤其是安全方面如应用间 隔离、权限管理方面，我们都不用像公有

云那样花大量精力去做，所以工作量其实还好。DAE现在在计划开源，当然它现在只支持Python应用。以后我们也 许会让DAE支持Go语言。

上面是在线的部分，对高可用性和低时延有较大要求。离线部分则包括数据挖掘、数据分析等，技术组件分别是海量分布式文件系统MooseFS，这个文 件系统的结构类似HDFS，用C语言编写，其好处在于FUSE模块实现的比较好，用文件系统就可以直接进行操作，而不需要专门的命令，可以支持的数据量也 很大。另外就是自己开发的分布式计算平台DPark。

DPark顾名思义是Spark的Python实现，不过现在已经跟Spark越来越不一样了。和 Hadoop 相比，Spark可以使用内存做为缓存加速分布式计算，DPark继承了这个优点，这对于大规模数据的迭代计算非常有用。在豆瓣的应用场景下，因为我们的 离线计算很多是推荐算法计算，这种计算涉及大量的迭代算法，如果每次计算的结果都入磁盘再在下一轮计算加载，那性能是很差的，所以DPark能够大幅提升 性能。另外，因为DPark的编写使用了函数式语言的特点，所以可以写的非常简洁：

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

```
import dpark

file = dpark.textFile("/mfs/words.txt")
words = file.flatMap(lambda x:x.split()
        ).map(lambda x:(x,1))
wc = words.reduceByKey(
        lambda x,y:x+y).collectAsMap()
print wc
```

到目前（2014年3月），DPark的集群规模和处理数据量已经比去年多了一倍左右，一天要处理60~100TB左右的数据。

团队

当前，我所负责的豆瓣平台部一共包括四个部分：核心系统，这块也是由我直接带领的，共6名工程师；DAE，现在是彭宇负责，共4名工程师；DBA两人；SA两人。

平台部负责的项目大多是跟业务无关的东西，贴近应用层的主要在产品线团队做，这个分工跟豆瓣工程团队的发展历史有关。早期豆瓣工程师还不多的时候， 就已经分为两种倾向，一种是偏业务的，就是去做用户能看得见的东西；另一种是支持性的，运行在业务层下面、不被用户所感知的东西。下面这一层就衍变成了平 台部门。

在豆瓣，不管是做产品还是做平台的工程师，技术实力都比较强，一个项目应该从哪个部门发起，并不是看这个任务的难度，而是看它是公共的还是业务特有 的。有些项目即使未来可能会成为公共的，但一开始只是一个产品线需要，那么它也会从产品线发起。比如豆瓣的短信服务，最开始是产品线有需求，所以这些服务 都是由他们发起完成的，平台这边主要负责提供建设服务的架构，比如DoubanService，告诉他们一个服务怎样去写、怎样去部署、怎样去对用户开 放。短信服务后来成为很多产品线都在使用的服务，同时这个系统本身也越来越成熟，那么它逐渐就被转移到SA团队来进行维护。

核心系统组做过的项目，包括刚才提到的DPark、BeansDB，还有MooseFS这些二次开发的，还有搜索服务、信息推送的长连接服务等，大大小小差不多有十几个。有些项目处于维护状态，所以需要的人不是那么多。

跟豆瓣其他工程团队一样，平台部也强制大家做code review。这对于核心系统来说很重要的一点在于，code review是一个知识共享的过程：我们人少项目多，所以很多项目都是一个人做主力，很容易就变成其他人不知道你这个项目具体是什么情况，而强制code review就可以实现一种公开透明的状态，让大家都了解每个项目在做什么。

在平台部，因为你做的所有东西都会影响到全公司，测试显然很重要，我们还做了另一件事来进行质量保证，那就是一个项目由谁来主导上线，谁就要负责这 个项目的故障响应——所有运维、调整系统等SA的工作，你这个第一负责人都要参与。你做的东西的好坏会影响到自己晚上能不能睡好觉，所以大家就会比较谨 慎。灰度上线也是我们这边的通用做法。

平台部还有一点跟产品线不一样的是，平台部没有产品经理，所以你的工作方向更多是自己去找的，每个人自己发现问题的能力更重要。我们每个月都会问大家，你这个月想要解决什么问题? 如果方向大家一致认可，那就去做。

最后，对于新技术的引入上，豆瓣整体是比较偏激进的，我们鼓励大家去看看新的技术。当然我们也不会看到新的就上，这里面有一些限制：一个是比较重要 的服务如果要上新的技术，一定要有成功案例，且成功案例有跟我们量级差不多的规模，这样可以降低风险；另一个是对于引入的新技术一定要吃透——大部分引入 的技术肯定是要做二次开发的，所以拿进来的技术你必须保证能完全理解它的代码结构，出了问题能修，能去掉自己无法掌控的东西。这也是为什么豆瓣不太可能在 重要的地方引入Java的原因，除非别无选择，我们一般都是Python、C和Go。

采访人介绍

庄表伟，目前在华为2012实验室的研发能力中心工作。也是80年代中期就开始接触计算机和编程，一直对软件开发、技术社区、开源软件等抱有浓厚的兴趣。近期关注的重点是：如何将开源社区的经验，应用于企业内部实践。

作者：洪强宁

本文转载自：http://www.infoq.com/cn/articles/douban-infrastructure-2014

从CODE REVIEW谈如何做技术

作者/陈皓
http://coolshell.cn/

（这篇文章缘由我的微博，我想多说一些，有些杂乱，想到哪写到哪）

这两天，在微博上表达了一下Code Rview的重要性。因为翻看了阿里内部的Review Board上的记录，从上面发现Code Review做得好的是一些比较偏技术的团队，而偏业务的技术团队基本上没有看到Code Rview的记录。当然，这并不能说没有记录他们就没有做Code Rview，于是，我就问了一下以前在业务团队做过的同事有没有Code Rview，他告诉我不但没有Code Review，而且他认为Code Review没用，因为：

1）工期压得太紧，时间连coding都不够，以上线为目的，

2）需求老变，代码的生命周期太短。所以，写好的代码没有任何意义，烂就烂吧，反正与绩效无关。

我心里非常不认同这样的观点，我觉得我是程序员，我是工程师，就像医生一样，不是把病人医好就好了，还要对病人的长期健康负责。对于常见病，要很快 地医好病人很简单，下猛药，大量使用抗生素，好得飞快。但大家都知道，这明显是“饮鸩止渴”、“竭泽而渔”的做法。医生需要有责任心和医德，我也觉得程序 员工程师也要有相应的责任心和相应的修养。东西交给我我必需要负责，我觉得这种负责和修养不是”做出来“就了事了，而是要到“做漂亮”这个级别，这就是 “山寨”和“工业”的差别。而只以“做出来”为目的标准，我只能以为，这样的做法只不过是“按部就班”的堆砌代码罢了，和劳动密集型的“装配生产线”和 “砌砖头”没有什么差别，在这种环境里呆着还不如离开。

老实说，因为去年我在业务团队的时候，我的团队也没有做Code Review，原因是多样的。其中一个重要原因是，我刚来阿里，所以，需要做的是在适应阿里的文化，任何公司都有自己的风格和特点，任何公司的做法都有他 的理由和成因，对于我这样的一个初来者，首要的是要适应和观察，不要对团队做太多的改动，跟从、理解和信任是融入的关键。（注：在建北京团队和不要专职的 测试人员上我都受到了一些阻力），所以跟着团队走没有玩Code Review。干了一年后，觉得我妥协了很多我以前所坚持的东西，觉得自己的标准在降低，想一想后背拔凉拔凉的，所以我决定坚持，而且还要坚持高标准。

对于Code Review很重要的这个观点，在微博上抛出来后，被一些阿里的工程师，架构师/专家，甚至资深架构师批评，我在和他们回复和讨论的过程中，居然发现有个 “因为对方用户的设置”我无法回复了（我被拉黑了，还有一些直接就是冷讽和骂人了，微博中我就直接删除了）。这些批评我的阿里工程师/架构师的观点总结一 下如下：（**顺便说一下，阿里内还是有很多团队坚持做Code Review的**）

1）到业务团队体会一下，倒逼工期的项目有多少？订好交付日期后再要求提前1个月的有多少？现在是做到已经不容易，更不谈做得漂亮！。

2）Code Review是一种教条，意义不大，有测试，只要不出错，就可以了。

3）目标都是改进质量，有限的投入总希望能有最大的产出，不同沉湎改进质量的方式不一样，业务应用开发忙的跟狗一样，而且业务逻辑变化快，通用性差，codereviw的成本要比底层高。

4）现在的主要矛盾是倒排出来的工期和不靠谱的程序员之间的矛盾，我认为cr不是解决这个问题的银弹。不从实际情况出发光打正义的嘴炮实在太过于自慰了 。

我们可以看到，上面观点其实和Code Rview没有太多关系，其实是在抱怨另外的问题。这些观点其实是技 术团队和业务团队的矛盾，但不知道为什么强加给了我的“Code Review很重要”的这个观点，然后这些观点反过来冲击“Code Reivew”，并说“Code Review无用”。这种讨论问题的方式在很常见，你说A，我说B，本来A、B是两件事，但就是要混为一谈，然后似是而非的用B来证明你的A观点是错的。 （也许，这些工程师/架构师心存怨气，需要一个发泄的通道）

我觉得，很多时候，人思考问题思考不清楚，很大一部分原因是因为把很多问题混为一谈，连我自己有些时候都会这样。引以为戒 。

即然被混为一谈，那我就来拆分一下，也是下面这三个问题：

- Code Review有没有用的问题。
- Code Review做不起来的问题。
- 业务变化快，速度快的问题，技术疲于跟命。

Code Review

你Google一下Code Reivew这个关键词，你就会发现Code Review的好处基本上是不存在争议的，有很多很多的文章和博文都在说Code Review的重要性，怎么做会更好，而且很多公司在面试过程中会加入“Code Review”的问题。打开Wikipedia的词条你会看到这样的描述——

卡珀斯·琼斯（*Capers Jones*）分析了超过*12,000*个软件开发项目，其中使用正式代码审查的项目，发现潜在缺陷率约在*60-65%*之间，若是非正式的代码审查，发现潜在缺陷率不到*50%*。大部份的测试，发现的潜在缺陷率会在*30%*左右。

对于一些关键的软件（例如安全关键系统的嵌入式软件），一般的代码审查速度约是一小时*150*行程序码，一小时审查数百行程序码的审查速度太快，可能无法找到程序中的问题。代码审查一般可以找到及移除约*65%*的错误，最高可以到*85%*。

也有研究针对代码审查找到的缺陷类型进行分析。代码审查找到的缺陷中，有*75%*是和计算机安全隐患有关。对于产品生命周期很长的软件公司而言，代码审查是很有效的工具。

Code Review的好处我觉得不用多说了，主要是让你的代码可以更好的组织起来，有更易读，有更高的维护性，同时可以达到知识共享，找到bug只是其中的副产品。这个东西已经不新鲜了，你上网可以找到很多文章，我就不多说了。就像你写程序要判断错误一样，Code Rview也是最基本的常识性的东西。

我从2002年开始就浸泡在严格的Code Review中，我的个人成长和Code Rview有很大的关系，如果我的成长过程中没有经历过Code Review这个事，我完全不敢想像。

我个人认为代码有这几种级别：1）可编译，2）可运行，3）可测试，4）可读，5）可维护，6）可重用。通过自动化测试的代码只能达到第3）级，而通过Code Rview的代码少会在第4）级甚至更高。关于Code Review，你可以参看本站的《Code Review中的几个提示》

可见，Code Review直接关系到了你的工程能力！

Code Review 的问题

有下面几个情况会让你的Code Review没有效果。

首当其冲的是——“**人员能力不足**”，我经历过这样的情况，Code Review的过程中，大家大眼瞪小眼，没有什么好的想法，不知道什么是好的代码，什么是不好的代码。导致Code Review大多数都在代码风格上。今天，我告诉你，代码风格这种事，是每个程序员自查的事情，不应该浪费大家的时间。对此，我有两个建议：1）你团队的 人招错了，该换血了。2）让你团队的人花时候阅读一下《代码大全》这本书（当然，还要读很多基础知识的书）。

次当其冲的是——“**结果更重要**”，也就是说，做出来更重要，做漂亮不重要。因为我的KPI和年终奖based on how many works I’ve done！而不是How perfect they are！这让我想到那些天天在用Spring MVC 做CRUD网页的工程师，我承认，他们很熟练。大量的重复劳动。其实，仔细想一下好多东西是可以框架化，模板化，或是自动生成的。所以，为了堆出这么多网 页就停地去堆，做的东西是很多，但是没有任何成长。急功近利，也许，你做得多，拿到了不错的年终奖，但是

你失去的也多，失去了成为一个卓越工程师的机会。 你本来可以让你的月薪在1-2年后翻1-2倍的，但一年后你只拿到了为数不多的年终奖。

然后是——“**人员的态度问题**”，一方面就是懒，不想精益求精，只要干完活交差了事。对此，你更要大力开展 Code Review了，让这种人写出来的代码曝光在更多人面前，让他为质量不好的代码蒙羞。另一方面，有人会觉得那是别人的模块，我不懂，也没时间 去懂，不懂他的业务怎么做Code Review? 我只想说，如果你的团队里这样的“各个自扫门前雪”的事越多，那么这个团队也就越没主动性，没有主动性也就越不可能是个好团队，做的东西也不可能好。而对 于个人来说，也就越不可能有成长。

接下来是——“**需求变化的问题**”，有人认识，需求变得快，代码的生存周期比较短，不需要好的代码，反正过两天这 些代码就会被废弃了。如果是一次性的东西，的确质量不需要太高，反正用了就扔。但是，我觉得多多少少要Review一下这个一次性的烂代码不会影响那些长 期在用的代码吧，如果你的项目全部都是临时代码，那么你团队是不是也是一个临时团队？关于如果应对需求变化，你可以看看本站的《需求变化与IoC》《Unix的设计思想来应对多变的需求》的文章 ，从这些文章中，我相信你可以看到对于需求变化的代码质量需要的更高。

最后是——“**时间不够问题**”，如果是业务逼得紧，让你疲于奔命，那么这不是Code Review好不好问题，这是需求管理和项目管理的问题以及别的非技术的问题。下面我会说。

不管怎么样，上述Code Review的问题不应该成为“Code Review无意义”的理由或借口，这就好像“因噎废食”一样。干什么事都会有困难和问题的，有的人就这样退缩了，但有的人看得到利大于弊，还是去坚持， 人与人的不同正在这个地方。这就是为什么运动会受伤，但还是会人去运动，而有人因为怕受伤就退缩了一样。

被业务逼得太紧

被业务逼得太紧，需求乱变，这其实和Code Rview没有多大关系了。对此，我想先讲一个我的故事。

我去年在阿里的聚石塔，刚去的时候，聚石塔正在做一个很大的重构——对架构的大调整。因此压了很多业务需求，等这个项目花了2-3个月做完了后，一 下子涌入了30-50个需求，还规定一个月完成，搞得团队疲于奔命。在累了两周后，我仔细分析了一下这些需求，发现很多需求是在重复做阿里云已经做过的东 西，还有一些需求是因为聚石塔这个平台不规范没有标准所产生的问题。于是，我做了这么三件事：

1）重新定义聚石塔这个产品主要目标和范围，确定哪些该做，哪些不该做。

2）为聚石塔制定标准 ，让阿里云的API都长得基本一样，并制订云资源的接入标准。

3）推动重构阿里云的Portal系统，不再实现阿里云已经做过的东西，与阿里云紧密结合。

这些事情推动起来并不容易，聚石塔的业务方一开始也不理解，我和产品一起做业务方的工作，而阿里云也被我逼得很惨（在这里一并感谢，尤其阿里云的同 学，老实说，和阿里云跨团队合作中是我这么多年来感觉最好的一次，相当赞）。通过这个事，聚石塔需求一下就有质的下降了。搞得还有几个工程师来和我说，你 这么搞，聚石塔就没事可干了。姑且不说工程师对聚石塔的理解是怎么样的。 我只想说，我大量地减少了需求，尽最大可能联合了该联合的人，而不是自己闭门造车，并让产品的目标和方向更明确了。做了这些事情后，大家不但不用加班，而 且还有时间充电去学技术，并为聚石塔思考未来的方向和发展。去年公司996的时候，我的团队还在965（搞得跟异教徒似的），而且还有很多时间去专研新的 东西。

说这个故事，我不是为了得瑟，而是因为有些人在微博上抨击我是一个道貌岸然的只会谈概念讲道理的装逼犯。所以，我告诉大家我在聚石塔是怎么做的，我 公开写在这里，你也可以向相关的同学去求证我说的是不是真的。也向你证明，我可能是个装逼犯，但绝不是只会谈概念讲道理的装逼犯。

被业务方逼得紧不要抱怨，你没有时间被逼得像牲口一样工作，这个时候，你需要的是暂停一下想一想，为什么会像牲口一样？而这正是让你变得聪明的机会。

我为你总结一下，

1）你有没有去Review业务部门给你的这么多的需求，哪些是合理的，哪些是不合理 的。在Amazon，开发工程师都会被教育拿到需求后一定要问——“为什么要做？业务影响度有多大？有多少用户受益？”，回答不清这个问题，没有数据的支 持，就不做。所以，产品经理要做很多数据挖拙和用户调研的工作，而不是拍拍脑袋，听极少数的用户抱怨就要开需求了。

2）产品经理也要管理和教育的。你要告诉你的产品经理：“你是一个好的产品经理，因为你 不但对用户把握得很好，也会对软件工艺把握得很好。你不但会开出外在的功能性需求，也同样会开出内在的让软件系统更完善的非功能性需求。你不是在迁就用 户，而是引导用户。你不会无限制地加功能，而是把握产品灵魂控制并简化功能。你会为自己要做的和不做东西的感到同样的自豪。”你要告诉你的产品经理：“做 一个半成品不如做好半年产品”（更多这样的观点请参看《Re-work摘录和感想》）

3）做事情是要讲效率的。Amazon里喜欢使用一种叫T-Shirt Size Estimation的评估方法来优先做投入小产出大的“Happy Case”。关于什么是效率，什么是T-Shirt Size Estimation，你可以看看《加班与效率》一文 。

4）需求总是会变化的，不要抱怨需求变化太快。你应该抱怨的是为什么我们没有把握好方 向？老变？这个事就像踢足球一样，你要去的地方是球将要去的地方，而不是球现在的地方。你要知道球要去哪里，你就知道球之前是怎么动的，找到了运动轨迹 后，你才知道球要去像何方。如果你都不知道球要去向何方，那你就是一只无头苍蝇一样，东一下西一下。

当你忙得跟牲口一样，你应该停下来，问一下自己，自己成为牲口的原因，是不是就是因为自己做事时候像就牲口一样思考?

其它

最后，我在给阿里今年新入职的毕业生的“技塑人生”的分享中，我给他们布置了5、6个Homework，分享几个给大家：

1）重构或写一个模块，把他做成真正的Elegant级别。

2）与大家分享一篇或几篇技术文章 ，并收获10-30个赞。

3）降低现有至少20%的重复工作或维护工作

4）拒绝或简化一个需求（需要项目中所有的Stakeholders都同意）

部署这些作业的原因，是我希望新入行的同学们对自己的工作坚持高的标准，我知道你们会因为骨感的现实而妥协，但是我希望你们就算在现实中妥协了也要在内心中坚持尽可能高的标准，不要习惯成自然，最后被社会这个大染缸给潜移默化了。因为你至少要对自己负责。**对自己负责就是，用脚投票，如果妥协得受不了了就离开吧。**

芝兰生于空谷，不以无人而不芳！君子修身养道，不以穷困而改志！

谢谢听我唠叨。

作者：@左耳朵耗子

原文链接：http://coolshell.cn/articles/11432.html

应用SCIKIT-LEARN做文本分类

作者/Rachel-Zhang
zju-cser;
http://blog.csdn.net/abcjennifer

文本挖掘的paper没找到统一的benchmark，只好自己跑程序，走过路过的前辈如果知道20newsgroups或者其它好用的公共数据集的分类（最好要所有类分类结果，全部或取部分特征无所谓）麻烦留言告知下现在的benchmark，万谢！

嗯，说正文。20newsgroups官网上给出了3个数据集，这里我们用最原始的20news-19997.tar.gz。

分为以下几个过程：

- 加载数据集
- 提feature
- 分类
  - Naive Bayes
  - KNN
  - SVM
- 聚类

说明： scipy官网上有参考，但是看着有点乱，而且有bug。本文中我们分块来看。

Environment：Python 2.7 + Scipy （scikit-learn）

1.加载数据集

从20news-19997.tar.gz下载数据集，解压到scikit_learn_data文件夹下，加载数据，详见code注释。

[python] view plaincopy

```
1. #first extract the 20 news_group dataset to /
   scikit_learn_data
2. from sklearn.datasets import fetch_20newsgroups

3. #all categories
4. #newsgroup_train = fetch_20newsgroups(subset='t
   rain')
5. #part categories
6. categories = ['comp.graphics',
7.  'comp.os.ms-windows.misc',
8.  'comp.sys.ibm.pc.hardware',
9.  'comp.sys.mac.hardware',
10. 'comp.windows.x'];
```

```
    newsgroup_train = fetch_20newsgroups(subset = 'train',categories = categorie
    s);
```

可以检验是否load好了：

[python] view plaincopy

```
1. #print category names
2. from pprint import pprint
3. pprint(list(newsgroup_train.target_names))
```

结果：

```
['comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x']
```

2. 提feature：

刚才load进来的newsgroup_train就是一篇篇document，我们要从中提取feature，即词频啊神马的，用fit_transform

[python] view plaincopy

```
1. #newsgroup_train.data is the original documents, but we need to extract the

2. #TF-IDF vectors inorder to model the text data
3. from sklearn.feature_extraction.text import TfidfVectorizer, HashingVectoriz
   er
4. #vectorizer = TfidfVectorizer(sublinear_tf = True,
5. #                             max_df = 0.5,
6. #                             stop_words = 'english');
7. #however, Tf-Idf feather extractor makes the training set and testing set ha
   ve
8. #divergent number of features. (Because they have different vocabulary in do
   cuments)
9. #So we use HashingVectorizer
10. vectorizer = HashingVectorizer(stop_words = 'english',non_negative = True,
11.                                n_features = 100)
12. fea_train = vectorizer.fit_transform(newsgroup_train.data)
13. #return feature vector 'fea_train' [n_samples,n_features]
14. print 'Size of fea_train:' + repr(fea_train.shape)
15. #11314 documents, 130107 vectors for all categories
```

```
print 'The average feature sparsity is {0:.3f}%'.format(
fea_train.nnz/float(fea_train.shape[0]*fea_train.shape[1])*100);
```

结果：

```
Size of fea_train:(2936, 100)
The average feature sparsity is 51.183%
```

因为我们只取了100个词，即100维feature，稀疏度还不算低。而实际上用TfidfVectorizer统计可得到上万维的feature，我统计的全部样本是13w多维，就是一个相当稀疏的矩阵了。

3. 分类

3.1 Multinomial Naive Bayes Classifier

见代码&comment，不解释

[python] view plaincopy

```
##########################################################
#Multinomial Naive Bayes Classifier
print '**************************\nNaive Bayes\n**************************'
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
newsgroups_test = fetch_20newsgroups(subset = 'test',
                                     categories = categories);
fea_test = vectorizer.fit_transform(newsgroups_test.data);
#create the Multinomial Naive Bayesian Classifier
clf = MultinomialNB(alpha = 0.01)
clf.fit(fea_train,newsgroup_train.target);
pred = clf.predict(fea_test);
calculate_result(newsgroups_test.target,pred);
#notice here we can see that f1_score is not equal to 2*precision*recall/(pr
ecision+recall)
#because the m_precision and m_recall we get is averaged, however, metrics.f
1_score() calculates
#weithed average, i.e., takes into the number of each class into considerati
on.
```

注意我最后的3行注释，为什么f1≠2*（准确率*召回率）/（准确率+召回率）

其中，函数calculate_result计算f1：

[python] view plaincopy

```
def calculate_result(actual,pred):
    m_precision = metrics.precision_score(actual,pred);
    m_recall = metrics.recall_score(actual,pred);
    print 'predict info:'
    print 'precision:{0:.3f}'.format(m_precision)
    print 'recall:{0:0.3f}'.format(m_recall);
    print 'f1-score:{0:.3f}'.format(metrics.f1_score(actual,pred));

```

3.2 KNN:

[python] view plaincopy

```
##############################################################
#KNN Classifier
from sklearn.neighbors import KNeighborsClassifier
print '************************\nKNN\n************************'
knnclf = KNeighborsClassifier()#default with k=5
knnclf.fit(fea_train,newsgroup_train.target)
pred = knnclf.predict(fea_test);
calculate_result(newsgroups_test.target,pred);
```

3.3 SVM:

[cpp] view plaincopy

```
##############################################################
#SVM Classifier
from sklearn.svm import SVC
print '************************\nSVM\n************************'
svclf = SVC(kernel = 'linear')#default with 'rbf'
svclf.fit(fea_train,newsgroup_train.target)
pred = svclf.predict(fea_test);
calculate_result(newsgroups_test.target,pred);
```

结果:

```
************************
Naive Bayes
************************
predict info:
precision:0.448
recall:0.448
```

```
f1-score:0.447
************************
KNN
************************
predict info:
precision:0.415
recall:0.405
f1-score:0.406
************************
SVM
************************
predict info:
precision:0.440
recall:0.438
f1-score:0.438
```

4. 聚类

5. [cpp] view plaincopy

```
1. ###########################################################
2. #KMeans Cluster
3. from sklearn.cluster import KMeans
4. print '************************\nKMeans\n************************'
5. pred = KMeans(n_clusters=5)
6. pred.fit(fea_test)
7. calculate_result(newsgroups_test.target,pred.labels_);
```

结果：

```
************************
KMeans
************************
predict info:
precision:0.177
recall:0.176
f1-score:0.171
```

本文全部代码下载：在此

作者：Rachel-Zhang

本文转载自：http://blog.csdn.net/abcjennifer/article/details/23615947

技术宅打造全能美剧播放器

作者/牙僧
非主流 程序猿 www.yaseng.me
高空工作室 www.uauc.net

1:前言

看到有同仁发《权力的游戏》自动追剧脚本,老衲也来凑个热闹…

移动端最好的播放器非云播君莫属了,极速而方便,高清而无码,可惜最近资源被和谐,每次有美剧更新需要用浏览器手工添加播放源或者pc端添加,很是 麻烦的说。于是就简单diy了下云播1.9 for Android ,使其支持自定义搜索引擎(比如人人影视,xxx资源站等),并且修复了原来搜索结果列表页标题的bug,为了迎接 Game of Thrones Season 4 的回归,特记录下修改过程。

2:添加yyets.com搜索引擎

反编译搜索引擎代码
com\xunlei\cloud\action\search\AdviseEngine.java 89 行

```
try
    {
      str = SearchAdviceEngineListResp.getDomainName
(paramString);
      if (str.equalsIgnoreCase("btdigg.org"))
      {
        localAdviseEngine.url_pattern = "http://btdi
gg.org/search?q={searchTerms}&p={page}";
        localAdviseEngine.multi_page = true;
        localAdviseEngine.page_start_point = 0;
```

```
        return localAdviseEngine;
      }
      if (str.equalsIgnoreCase("so.com"))
      {
        localAdviseEngine.url_pattern = "http://www.so.com/s?q={searchTerms}+sit
e%3Abtdigg.org&pn={page}";
        localAdviseEngine.multi_page = true;
        localAdviseEngine.page_start_point = 1;
        return localAdviseEngine;
      }
    }
    catch (URISyntaxException localURISyntaxException)
    {
      localURISyntaxException.printStackTrace();
      return localAdviseEngine;
    }
    if (str.equalsIgnoreCase("torrentkitty.com"))
    {
      localAdviseEngine.url_pattern = "http://www.torrentkitty.com/search/{searc
hTerms}/{page}";
      localAdviseEngine.multi_page = true;
      localAdviseEngine.page_start_point = 1;
      return localAdviseEngine;
    }
    if (str.equalsIgnoreCase("kuaichuanmirror.com"))
    {
      localAdviseEngine.url_pattern = "http://www.kuaichuanmirror.com/search/{se
archTerms}/";
      localAdviseEngine.multi_page = false;
      localAdviseEngine.page_start_point = 1;
    }
```

挖槽 竟然内置四大毛片搜索引擎 !!!
随便改掉一个即可
由于内置引擎只识别磁力链接 需要一个脚本文件来做中转(代码见最后链接)
输出格式为

```
magnet:?xt=urn:btih:hash1&title1</br>
magnet:?xt=urn:btih:hash2&title2</br>
```

修改 smali\com\xunlei\cloud\action\search\AdviseEngine.smali 239 行

```
.line 127
:cond_1
const-string v2, "yyets.com"
invoke-virtual {v0, v2}, Ljava/lang/String;->equalsIgnoreCase(Ljava/lang/String;
)Z
move-result v2
```

```
if-eqz v2, :cond_2
.line 128
const-string v0, "http://www.ttst.com/y2c.php?keyword={searchTerms}&page={page}"

iput-object v0, v1, Lcom/xunlei/cloud/action/search/AdviseEngine;->url_pattern:L
java/lang/String;
.line 129
const/4 v0, 0x1
```

3:修正标题bug

云播在使用第三方搜索引擎时,发现结果列表页面的标题全部是随机的字符串

查看对应的dalvik代码

smali\com\xunlei\cloud\action\search\AdviseEngine.smali 990行

```
const-string v0, "magnet:\\?xt=urn:btih:([A-Za-z\\d]{32,})&?"      //批量正则  遍历结果数组
  .....
 invoke-static {v0, v1}, Ljava/util/regex/Pattern;->compile(Ljava/lang/String;I)Ljava/util/regex/Pattern;
 .....
 invoke-virtual {v5}, Ljava/util/regex/Matcher;->find()Z
move-result-object v0
 .line 88
 invoke-virtual {v0}, Ljava/lang/String;->length()I
move-result v1
add-int/lit8 v1, v1, -0x1
 invoke-virtual {v0, v1}, Ljava/lang/String;->charAt(I)C
move-result v1
.line 89
const/16 v6, 0x26
if-ne v1, v6, :cond_4
.line 90
invoke-virtual {v0}, Ljava/lang/String;->length()I
move-result v1
add-int/lit8 v1, v1, -0x1
invoke-virtual {v0, v3, v1}, Ljava/lang/String;->substring(II)Ljava/lang/String;
 move-result-object v0
    move-object v1, v0
    .line 92
    :goto_1
    const/16 v0, 0x14
    invoke-virtual {v1}, Ljava/lang/String;->length()I
    move-result v6
    invoke-virtual {v1, v0, v6}, Ljava/lang/String;->substring(II)Ljava/lang/String;
    move-result-object v0
    .line 93
    invoke-static
 {v0, v2},
Lcom/xunlei/cloud/action/search/g;->a(Ljava/lang/String;I)Ljava/lang/String;
 // str2.substring(20, str2.length())
```

```
    move-result-object v6
 ......
.line 105
iput-object v6, v0, Lcom/xunlei/cloud/action/search/SnifferData;->title:Ljava/la
ng/String;
.line 106
 iput-object v1, v0, Lcom/xunlei/cloud/action/search/SnifferData;->url:Ljava/lan
g/String;
.line 107
 iput-object v6, v0, Lcom/xunlei/cloud/action/search/SnifferData;->hash:Ljava/la
ng/String;
```

可以看到,标题被直接赋值给 magnet 的 hash 了 .....

修改为

```
ArrayList localArrayList = new ArrayList();
  Matcher localMatcher = Pattern.compile(".*?</br>", 2).matcher(    //中转脚本每行用
</br> 分割
    paramString);
  if (!localMatcher.find()) {
   return;
  }
  String str1;
  str1 = localMatcher.group();
  for (String str2 = str1.substring(0, 60);; str2 = str1) {
   String str3 = str2.substring(20, 60);
   String str4 = str1.substring(61, str1.length() - 5);
   SnifferData localSnifferData = new SnifferData();
   localSnifferData.title = str4;
   localSnifferData.url = str2;
   localSnifferData.hash = str3;
   localArrayList.add(localSnifferData);
   break;
  }
```

对应的dalvik 代码(具体见后面链接)

```
locals 15 //15个寄存器
const-string v0, ".*?</br>"
const/4 v10, 0x0
const/16 v11, 0x3c // 60
const/16 v12, 0x3d //61
const/4 v1, 0x2
invoke-static {v0, v1}, Ljava/util/regex/Pattern;->compile(Ljava/lang/String;I)L
java/util/regex/Pattern;
move-result-object v0
invoke-virtual {v0, v10, v11}, Ljava/lang/String;->substring(II)Ljava/lang/Strin
g;
```

```
move-result-object v8
move-object v1, v8

.line 92
:goto_1
move-object v9, v0     // copy 一份v0   magnet:?xt=urn:btih:hash&title
const/16 v0, 0x14


invoke-virtual {v1}, Ljava/lang/String;->length()I


........
new-instance v0, Lcom/xunlei/cloud/action/search/SnifferData;
invoke-direct {v0}, Lcom/xunlei/cloud/action/search/SnifferData;-><init>()V
invoke-virtual {v9}, Ljava/lang/String;->length()I
move-result v13
add-int/lit8 v13, v13, -0x5
invoke-virtual {v9, v12, v13}, Ljava/lang/String;->substring(II)Ljava/lang/Strin
g;
move-result-object v14
.line 105
iput-object v14, v0, Lcom/xunlei/cloud/action/search/SnifferData;->title:Ljava/l
ang/String;


.line 106
iput-object v1, v0, Lcom/xunlei/cloud/action/search/SnifferData;->url:Ljava/lang
/String;


.line 107
iput-object v6, v0, Lcom/xunlei/cloud/action/search/SnifferData;->hash:Ljava/lan
g/String;
```
4:回编译错误处理

```
java -jar apktool.jar b CloudPlay_ybappgw c1.apk
libpng error: Not a PNG file
ERROR: Failure processing PNG image F:\Pentest\Ya
Tools\pendroid\apktool1.5.2\CloudPlay_ybappgw\res\drawable-hdpi\local_list_view_
item_color.png
```
不是真正的png文件 看了下文件头 bmp... 改过来 继续

```
ERROR: 9-patch image F:\Pentest\Ya
```

```
Tools\pendroid\apktool1.5.2\CloudPlay_ybappgw\res\drawable-hdpi\progressbar_loca
l_item.9.png
malformed.
```
把三个文件全部改成普通 png格式 progressbar_local_item.9.png => progressbar_local_item.png

```
apktool b CloudPlay_ybappgw c1.apk
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building apk file...
```
成功编译成apk 文件

5:测试结果

签名 安装到手机

```
java -jar signapk.jar testkey.x509.pem testkey.pk8 c1.apk c1_s.apk
adb install -r c1_s.apk
```
截图

6:链接

中转脚本: https://github.com/yaseng/pentest/blob/master/project/diy_cloudplay/y2c.php
g.smali : https://github.com/yaseng/pentest/blob/master/project/diy_cloudplay/g.smali
smali 语法 : https://code.google.com/p/smali/w/list
Av Top 10: http://movie.douban.com/doulist/3822549

作者：Yaseng

本文转载自：http://www.freebuf.com/articles/others-articles/31576.html

关于OPENSSL“心脏出血”漏洞的分析

译者/乌云知识库
天天学习，好好向上!
http://drops.wooyun.org

0x00 背景

当我分析GnuTLS的漏洞的时候，我曾经说过，那不会是我们看到的最后一个TLS栈上的严重bug。然而我没想到这次OpenSSL的bug会如此严重。

OpenSSL“心脏出血”漏洞是一个非常严重的问题。这个漏洞使攻击者能够从内存中读取多达64 KB的数据。一些安全研究员表示：

无需任何特权信息或身份验证，我们就可以从我们自己的（测试机上）偷来X.509证书的私钥、用户名与密码、聊天工具的消息、电子邮件以及重要的商业文档和通信等数据。

这一切是如何发生的呢？让我们一起从代码中一探究竟吧。

0x01 Bug

请看ssl/dl_both.c，漏洞的补丁从这行语句开始：

```
int
dtls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum pad-
ding */
```

一上来我们就拿到了一个指向一条SSLv3记录中数据的指针。结构体SSL3_RECORD的定义如下（译者注：结构体SSL3_RECORD不是SSLv3记录的实际存储格式。一条SSLv3记录所遵循的存储格式请参见下文分析）：

```
typedef struct ssl3_record_st
    {
        int type;               /* type of record */
        unsigned int length;    /* How many bytes
available */
        unsigned int off;       /* read/write offset
into 'buf' */
        unsigned char *data;    /* pointer to the
record data */
        unsigned char *input;   /* where the decode
bytes are */
```

```
        unsigned char *comp;    /* only used with decompression - malloc()ed */
        unsigned long epoch;    /* epoch number, needed by DTLS1 */
        unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
    } SSL3_RECORD;
```

每条SSLv3记录中包含一个类型域（type）、一个长度域（length）和一个指向记录数据的指针（data）。我们回头去看dtls1_process_heartbeat：

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;
```

SSLv3记录的第一个字节标明了心跳包的类型。宏n2s从指针p指向的数组中取出前两个字节，并把它们存入变量payload中——这实际上是心 跳包载荷的长度域（length）。注意程序并没有检查这条SSLv3记录的实际长度。变量pl则指向由访问者提供的心跳包数据。

这个函数的后面进行了以下工作：

```
unsigned char *buffer, *bp;
int r;

/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;
```

所以程序将分配一段由访问者指定大小的内存区域，这段内存区域最大为 (65535 + 1 + 2 + 16) 个字节。变量bp是用来访问这段内存区域的指针。

```
/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
```

宏s2n与宏n2s干的事情正好相反：s2n读入一个16 bit长的值，然后将它存成双字节值，所以s2n会将与请求的心跳包载荷长度相同的长度值存入变量payload。然后程序从pl处开始复制payload个字节到新分配的bp数组中——pl指向了用户提供的心跳包数据。最后，程序将所有数据发回给用户。那么Bug在哪里呢？

0x01a 用户可以控制变量payload和pl

如果用户并没有在心跳包中提供足够多的数据，会导致什么问题？比如pl指向的数据实际上只有一个字节，那么memcpy会把这条SSLv3记录之后的数据——无论那些数据是什么——都复制出来。

很明显，SSLv3记录附近有不少东西。

说实话，我对发现了OpenSSL“心脏出血”漏洞的那些人的声明感到吃惊。当我听到他们的声明时，我认为64 KB数据根本不足以推算出像私钥一类的数据。至少在x86上，堆是向高地址增长的，所以我认为对指针pl的读取只能读到新分配的内存区域，例如指针bp指 向的区域。存储私钥和其它信息的内存区域的分配早于对指针pl指向的内存区域的分配，所以攻击者是无法读到那些敏感数据的。当然，考虑到现代malloc 的各种神奇实现，我的推断并不总是成立的。

当然，你也没办法读取其它进程的数据，所以“重要的商业文档”必须位于当前进程的内存区域中、小于64 KB，并且刚好位于指针pl指向的内存块附近。

研究者声称他们成功恢复了密钥，我希望能看到PoC。如果你找到了PoC，请联系我。

0x01b 漏洞修补

修复代码中最重要的一部分如下：

```
/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

这段代码干了两件事情：首先第一行语句抛弃了长度为0的心跳包，然后第二步检查确保了心跳包足够长。就这么简单。

0x02 前车之鉴

我们能从这个漏洞中学到什么呢?

我是C的粉丝。这是我最早接触的编程语言，也是我在工作中使用的第一门得心应手的语言。但是和之前相比，现在我更清楚地看到了C语言的局限性。

从GnuTLS漏洞和这个漏洞出发，我认为我们应当做到下面三条：

花钱请人对像OpenSSL这样的关键安全基础设施进行安全审计；
为这些库写大量的单元测试和综合测试；
开始在更安全的语言中编写替代品。
考虑到使用C语言进行安全编程的困难性，我不认为还有什么其他的解决方案。我会试着做这些，你呢?

作者简介：Sean是一位关于如何把事儿干好的软件工程师。现在他在Squadron工作。Squadron是一个专为SaaS应用程序准备的配置与发布管理工具。

测试版本的结果以及检测工具：

```
OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable
OpenSSL 1.0.1g is NOT vulnerable
OpenSSL 1.0.0 branch is NOT vulnerable
OpenSSL 0.9.8 branch is NOT vulnerable
```

http://filippo.io/Heartbleed/

原作者：Sean Cassidy 原作者Twitter: @ex509 原作者博客：http://blog.existentialize.com

来源：http://blog.existentialize.com/diagnosis-of-the-openssl-heartbleed-bug.html

本文转载自：http://drops.wooyun.org/papers/1381

层展现象

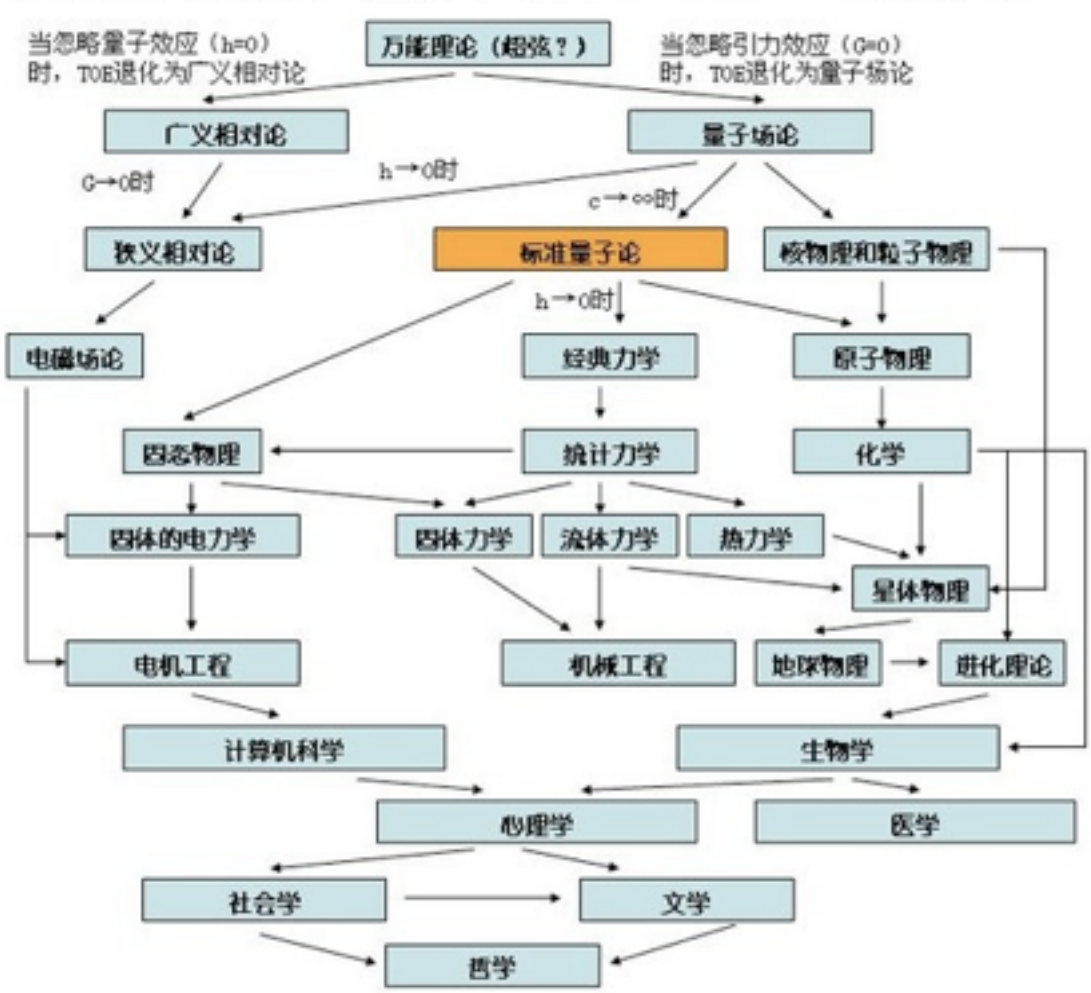
作者/lifesinger
岁月如歌

「知乎」是个好地方，经常会去逛一逛。今天在知乎上读到一个话题：「基础物理研究真的是整个科学发展的根基吗？」

这问题在学生时代也困扰过。自己从事的是基础物理研究，因此特别喜欢「基础物理是根基」的说法。人呀，都如此。都会在某些阶段，特别想证明、强调自己所从事工作的重要性。

还原论

「基础物理是一切自然学科根基」的说法，可以用下面这张未必正确的图来说明：



根基说法也叫「还原论」，即一切自然科学，都可以还原成某个基础学科。比如上图中，各种学科的起点，都是「万能理论」。

「还原论」的想法，为自然科学的发展做出了重要贡献，特别是对物理学本身的发展。

层展论

但是越来越多的现象表明，「还原论」未必能解释所有问题，即便我们拥有了「还原论」的终极梦想「万能理论」，依旧有很多很多自然现象无法解释。

引一段 L.P.Kadanoff 的讲话：

我在这里要反对还原论的偏见，我认为已经有相当的经验表明物质结构有不同的层次，而这些不同层次构成不同群落的科学家研究的领域，有一些人研究夸 克，另外一些人研究原子核，还有的研究原子、分子生物学，遗传学，在这个清单中，后面的部分是由前面部分构成的，每一个层次可以看成比它前面的好像低一 些，但每一个层次都有新的、激动人心的、有效的、普遍的规律，这些规律往往不能从所谓更基本的规律推导出来。从最不基本的问题向后倒推，我们可以看到一些 重要的科学成果。像门德尔的遗传律与 DNA 的双螺旋结构，量子力学与核裂变，谁是最基本的？谁推导谁？要将科学上的层次分高低的话，往往是愚蠢的，在每一层次上都有的普遍原则中，都会出现宏伟的概 念。

「层展论」的核心观念，可以用凝聚态理论学家 P.W.Anderson 的一段话来概括：

将一切事物还原成简单的基本规律的能力，并不意味着我们有能力从这些规律来重建宇宙，当面对尺度与复杂性的双重困难时，构筑论的假设就被破坏了。大量的复杂的基本粒子的集体，并不等于几个粒子性质的简单外推。

这与凯文·凯利在『失控』一书中的观点非常相近：无论对单个蜜蜂的研究多么深入，都无法解释蜂群的行为。

以上是回忆，回忆的背后，是对现状与未来的思考。

程序员心中的底层梦

我感知到的很多程序员，心中都或多或少有一些底层梦：想深入研究计算机的基础知识，比如编译原理、网络协议、硬件驱动等等。

这没什么不对，对底层的必要了解非常有必要。

然而，会看到程序员圈子里，也有很强的「还原论」群体。在这批人眼中，只要把基础知识深深掌握好后，一切其他上层语言、方案等都是手到擒来、小菜一碟。

程序员圈子里的「还原论」，大部分情况下都是对的。

但就如经典力学头上的两朵乌云一样，随着 IT 产业的迅猛发展，「还原论」头上已经飘出了很多乌云。

很多兴趣广泛的传统程序员，除了把 C++ / Java 等钻研得非常精透，也会跟随潮流，开始学习前端开发技能，比如 JavaScript / CSS / HTML 等。但是真让他们去写一个页面，经常只能差强人意，很难做得很好。

这究竟是为什么?

回到前面讨论的「展层论」，一切变得非常容易解释：

计算机学科也是分层的，上层部分需要基于底层构建，但每一层都有新的、激动人心的、有效的、普遍的规律，这些规律往往不能从所谓的基本规律推导出来。

这应该是一个常识，但我们却经常没有看到。各行各业皆如此。分层次没错，但给层次分高低，则往往是愚蠢而狂妄的。

最后

这篇文章最后的结论是常识，或者说是基本观念，但真的对吗？类似的，还有因果律。无论是自然学科，还是佛学，都赞同任何一种现象或事物都必有其因。

「层展论」、「因果律」等等观念，究竟是怎么回事？是否这些观念本身，只是人类愚昧且狂妄的投影?

你是怎么想的?

题图：那璀璨的星空深处，是否有我的爱?

作者：@玉伯也叫射雕

原文链接：https://github.com/lifesinger/lifesinger.github.com/issues/154

OPENSSL 的 HEARTBLEED 漏洞的影响到底有多大?

作者/余弦
知道创宇
《Web前端黑客技术揭秘》

写在前面

一切权威看爆这次漏洞的官方动态：Heartbleed Bug（Heartbleed这个命名不错，载入史册）。

@知道创宇安全研究团队 实测可以Dump出淘宝、微信、陌陌、某些支付类接口、某些比特币平台、12306等各大使用OpenSSL服务的一些内存信息，里面有用户等的敏感内容（有些重要网站含明文密码）。

OpenSSL这次被比做「心脏出血」。可见影响。一个安全套件不安全了……

下图的科普可以让大众明白这个OpenSSL漏洞是怎么回事：

权威统计

而且还不知是否有更进一步影响（小道八卦影响比想象的严重）。来自Heartbleed的官方说明（大概翻译下）：
OpenSSL 在Web容器如Apache/Nginx中使用，这两的全球份额超过66%。还在邮件服务如SMTP/POP/IMAP协议中使用，聊天服务如XMPP协 议，VPN服务等多种网络服务中广泛使用。幸运的是，这些服务很多比较古老，没更新到新的OpenSSL，所以不受影响，不过还是有很多用的是新的 OpenSSL，都受影响！
特别说明下：现在大家的聚焦都在HTTPS网站（443端口），实际上还有更多敏感服务受影响，我们的研究也是在不断持续推进！不可草率判断！！

1. HTTPS服务（443端口）

来自@ZoomEye 的统计（4.8号）：
全国443端口：1601250，有33303个受本次OpenSSL漏洞影响！注意这只是443端口。
全球443端口，至少受影响有71万量级（实际应该大于这个，待修正）。
ZoomEye OpenSSL漏洞国内的趋势监控（随时更新，仅是443端口）：
第一天：33303 台服务器（说明：国内是4.8号爆发的，当天的影响服务器情况！）
第二天：22611 台服务器（说明：辛苦一线白帽子与运维人员等的辛苦熬夜，减了1/3！而且都是知名业务，如百度、360、微信、陌陌、淘宝等，这点非常赞！）
第三天：17850 台服务器（说明：又减少了近500台服务器，我估计剩下的都不那么大，不过不排除有很重要的！）
第四天：15661 台服务器（说明：相比第一天减少了1/2！不过减少趋势慢了……）
第五天：14401 台服务器（说明：减少趋势持续缓慢……）
第六天：13854 台服务器（说明：减少趋势持续缓慢……）

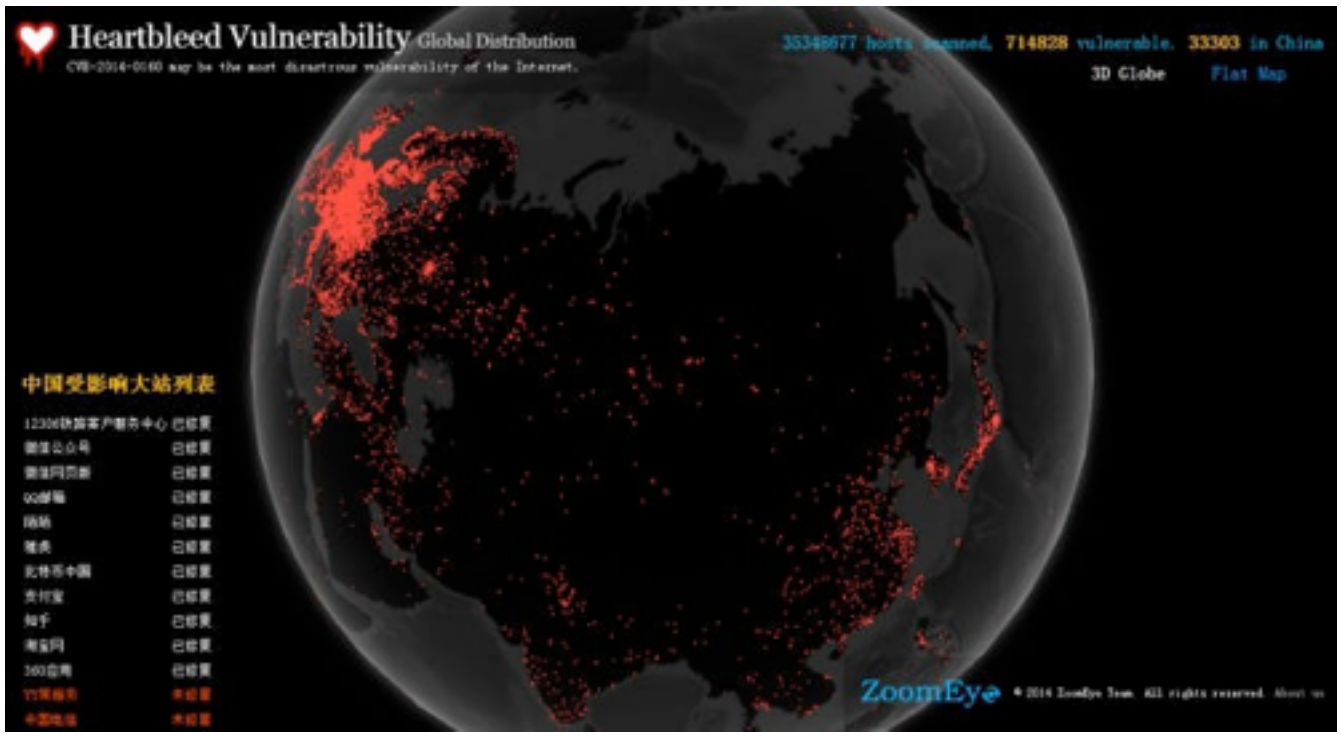
老外有个基于全球TOP1000的粗暴根域OpenSSL探测列表，仅供参考：
heartbleed-masstest/top1000.txt at master 路 musalbas/heartbleed-masstest 路 GitHub
说这个粗暴是因为：仅对「根域」。

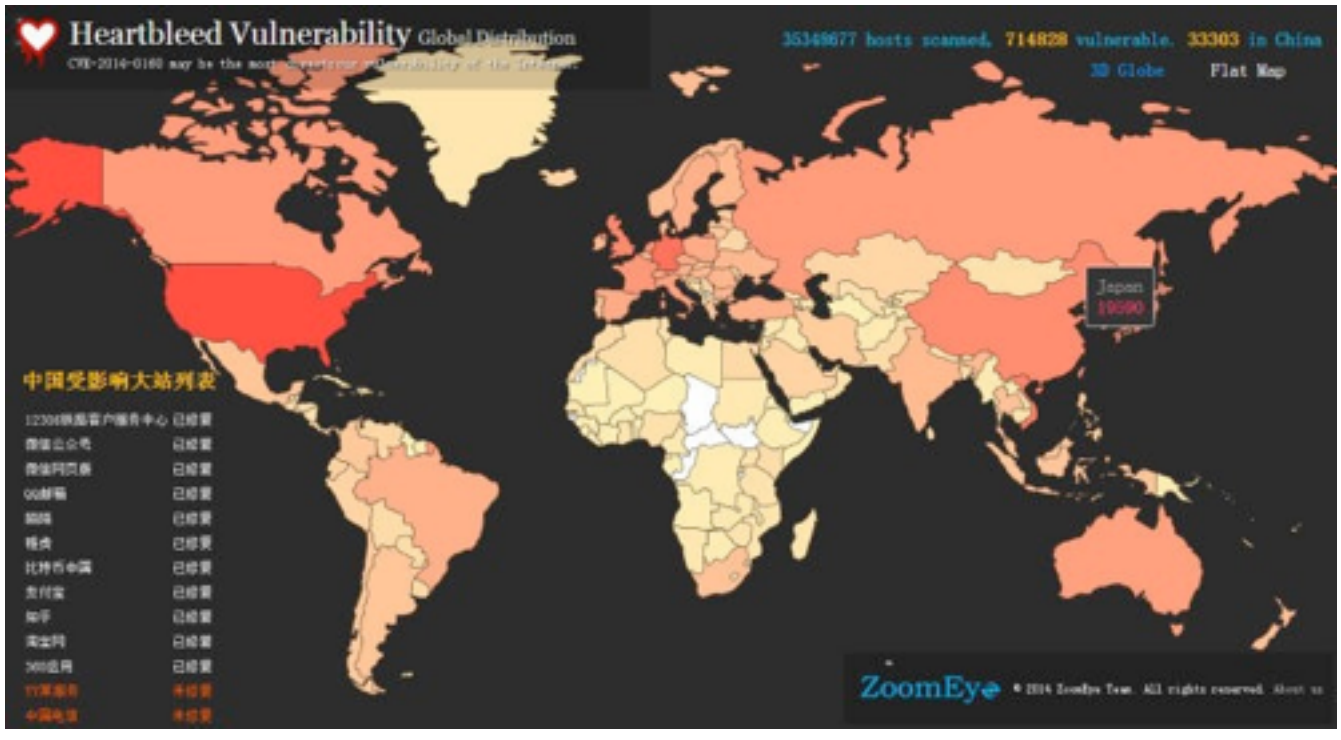
2. 邮件服务

来自@ZoomEye 的统计（4.11号）：
最新全球受影响服务及主机 SMTP Over SSL：147292台；POP3 over SSL：329747台；IMAP over SSL：353310台。
实测可以获取某些邮件服务的敏感数据。

3. 可视化

大家可以关注ZoomEye专门针对OpenSSL漏洞制作的全球监控页面（在Chrome或基于Webkit内核的浏览器下，效果最佳，目前仅是443端口）：
OpenSSL Heartbleed Worldwide Vulnerable Distribution
我们未来会持续完善这个页面，给大家一个专业、公正的评判结果，辛苦ZoomEye团队的几个小伙伴辛苦突击！截图两张：





这个漏洞的全球趋势图，会在一周后数据量足够的情况下给出。

疯掉

今晚（4.8号）不知道有多少团队要熬夜：

- 甲方，加急升级OpenSSL（如果升级真的可以的话，目前来看是可以）；
- 乙方，像我们这样的安全公司，在给我们全线产品+客户提供安全应急，并给社会输出有价值的参考信息；
- 地下黑客，刷库，各种刷！！！
- 每次这种大事，乌云都是被各路白帽子刷分的：

2014-04-08	京东某分站openssl漏洞导致敏感信息泄露及全站随机用户登录(证明可登录)
2014-04-08	豌豆荚运维不当导致服务器敏感信息泄露
2014-04-08	开源中国运维不当导致可随机登录用户并获取服务器敏感信息（成功登陆）
2014-04-08	okcoin运维不当导致可随机登录用户并获取服务器敏感信息
2014-04-08	LastPass运维不当导致敏感信息泄漏漏洞
2014-04-08	网易126邮箱运维不当导致可获取服务器敏感信息
2014-04-08	搜狗通行证服务器运维不当导致信息泄露
2014-04-08	唯品会运维不当导致敏感信息泄漏
2014-04-08	苏宁易购主站运维不当导致可以登录随机用户并且获取服务器敏感信息
2014-04-08	蘑菇街主站运维不当导致可以登录随机用户并且获取服务器敏感信息
2014-04-08	163邮箱运维不当导致可随机登录用户并获取服务器敏感信息
2014-04-08	中国银联运维不当导致可能存在随机登录银联账户并获取服务器敏感信息
2014-04-08	盛大网络运维不当导致敏感信息泄漏
2014-04-08	OWASP主站运维不当导致可获取服务器敏感信息
2014-04-08	360某api运维不当导致敏感信息泄漏
2014-04-08	雅虎主站运维不当导致可以登录随机用户并且获取服务器敏感信息
2014-04-08	微信网页版和公众账号版运维不当导致可随机登录微信用户并获取服务器敏感信息
2014-04-08	比特币中国运维不当导致随机用户明文密码泄漏
2014-04-08	12306新版订票系统运维不当导致可以登录随机用户并且获取服务器敏感信息

我们已经联合国家有关单位进行应急响应，我相信这次风波会很快平息。

用户防御建议

对于普通用户来说只要发现浏览器地址栏的网址是https开头的都应该警惕，因为这次OpenSSL漏洞影响的正是https网站，本来是安全传输的却也不安全了。可惜的是普通用户来说，有时候很难发现所使用的服务背后是https，比如：
有的仅在登录过程会https，之后都是http，典型的如百度的登录；
如果是手机上的APP等，普通用户就更不知道背后是不是https链接；

4.8号：和 某银行朋友交流，他们更新这个漏洞，需要2天时间！！这段时间，用户谨慎吧，不登录就不登录（因为你登录了你的相关明文信息，如用户名密码会在那万恶的 64K内存中，这段内存是可以被OpenSSL这个漏洞刷出来的……），等过3天或这些网站说修复了，大家再继续使用服务……

如果已经登录过，你紧张的话，就修改密码吧。

4.10号：最新的消息是「腾讯电脑管家」联合「安全联盟」发布了针对用户的解决方案：
做了两件事情：
一方面对50万个热门的网站扫描是否存在漏洞；
一方面会在管家用户访问https站点时，云端确认这个站点是否存在漏洞。如果用户访问有漏洞的站点，就出拦截页面提示用户，建议不要登录。
这个非常赞！

4.12号：那些知名的大网站，现在去修改密码会靠谱很多，因为这些大网站几乎修复完全了。

厂家防御建议

首先，这类攻击日志据Heartbleed官方说，无日志记录，追查不到。不过IDS/IPS等可以检测/防御（我们的加速乐也可以）。

别犹豫了！！尽快升级OpenSSL吧！！

如果厂家负责的话，可以学国外知名云平台厂家Heroku的做法：
Heroku | OpenSSL Heartbleed Security Update
升级后，提醒用户更改密码、提醒云服务使用者更新SSL密钥重复证书。不过不太指望国内厂家这样做，因为我相信用户绝对会疯掉。

附录参考

0. Heartbleed Bug（爆这次漏洞的官方）
1. ZoomEye团队的：OpenSSL Heartbleed Worldwide Vulnerable Distribution
2. 关于OpenSSL“心脏出血”漏洞的分析 - 乌云知识库 - 知乎专栏
3. @Evilm0 OpenSSL漏洞爆发后 - 微信公众号：Evil-say - 知乎专栏

这事的影响超乎想象，随时更新。

作者：知道创宇 余弦
本文转载自：http://www.zhihu.com/question/23328658/answer/24241031

程序猿语录

作者/恐龙丹佛
恐龙丹佛的 Geek 视野

1、栈和队列的区别是啥？ 吃多了拉就是队列；吃多了吐就是栈

2、世界上最遥远的距离不是生与死，而是你亲手制造的BUG就在你眼前，你却怎么都找不到她。

3、《c++程序设计语言》比《c程序设计语言》厚了几倍。。。果然有了对象就麻烦很多……

4、怎么使用面向对象的方式变得富有？继承。

5、为什么程序员总是分不清万圣节和圣诞节？因为 Oct 31 == Dec 25。

6、Keyboard not found … press F1 to continue

7、提交代码不写注释的人，小JJ就跟注释一样长

8、杀一个程序员不需要用枪，改三次需求就可以了

9、服务器按功能可以分为：数据库服务器，web服务器，cache服务器，下片儿服务器等等。

10、四个2B青年掐架。
A：你丫等着，我爹是敏感词！
B：Cao你大爷！你丫牛B神马，我爹在网上搜索根本无法显示！！
C： 我爹404 not found！！！
D：我爹Connection Reset！！！！

11、一同学问我，软件外包是什么。解释了几句还没明白，遂想了一下：包工头知道吧？顿悟！

12、十行代码九个警告八个错误竟然敢说七日精通六天学会五湖四海也不见如此三心二意之项目经理简直一等下流。

13、一个程序猿在肉店买了1公斤肉，回家一称，他不高兴的跑回肉店对老板说：少了24克……

14、网络聊天的时候，想表达对方是猪，一般人会打“xxx你这个猪。”，程序员会打 xxx.isPig = TRUE。

15、两个程序员，一个技术精湛，思维严谨，认真负责，Bug极少，至今单身； 一个技术一般，吊儿郎当，Bug一堆，经常被测试MM叫到她旁边，接受批评，后来成了她男朋友。

16、宝宝数学很好，2岁就可以从1数到10了。后来，我告诉他0比1还小。
今天吃饺子，我说：“宝宝，你数数你想吃几个饺子？”
“0，1，2，3。”一边说着一边拿起一个饺子，“这是第0个。”
老婆怒吼：“下一代还是做程序员的命！”

17、程序员找不到对象，一般有三种情况：
1. C#、JAVA都有对象，但是经常找不到对象。
2.ASM C直接没有对象。
3.javascript都是伪对象，最多算暧昧。
但C++日子一直都好过，因为C++是多继承，富二代呀！！！

18、程序猿：我的第一个问题是，对于我第二个和第三个问题，你可不可以只用‘能’和‘不能’来回答?
老板：“OK！”
我的第二个问题是，如果我的第三个问题是我能不能涨工资？那么你对于我的第三个问题的答案能不能和第二个问题的答案一样?
老板：。。。。。。

19、假如生活欺骗了你，找50个程序员问问为什么编程；
假如生活让你想死，找50个程序员问问BUG改完了没有；
假如你觉得生活拮据，找50个程序员问问工资涨了没有；
假如你觉得活着无聊，找50个程序员问问他们一天都干了什么！

20、c程序员看不起c++程序员， c++程序员看不起java程序员， java程序员看不起c#程序员，c#程序员看不起美工， 周末了，美工带着妹子出去约会了…一群傻X程序员还在加班！

21、客户被绑，蒙眼，惊问：“想干什么？”
对方不语，鞭答之，客户求饶：“别打，要钱？”
又一鞭，“十万够不？”
又一鞭，“一百万？”
又一鞭。客户崩溃：“你们TMD到底要啥？”
“要什么？我帮你做项目，写代码的时候也很想知道你TMD到底想要啥！”

22、“这位同学，请问你知道《边城》吗？”“呸！别跟我提编程，老子这辈子最讨厌的就是编程！”

23、摘自雷登书屋数据处理字典： 死循环: n.,见无限循环。 无限循环：n.,见死循环。

24、一个人正吸着雪茄，吐着烟圈。他女朋友生气了发飙道，“你没看见包装盒上的警告么？吸烟有害健康！”那人回答道：“我是程序员。我们不关心警告，只关心错误。”

作者: @恐龙丹佛

本文转载于：http://denveryao.com/html/programmer-jokes.html

【开源专访】谢宝友：会说话的LINUX内核

我们本次开源专访的对象是一位认真钻研技术的工程师，谢宝友，目前任职中兴通讯操作系统团队。他个人在业余时间前后共花费了6年时间完成了对Linux内核Linux 2.6.12内核源代码注释工作。近一个月之前，谢宝友（@kernel-hacker）发布的一条关于Linux 2.6.12内核源代码注释文件的微博，被转载近200次。该微博为大家介绍了包括内存、调度、文件系统等模块在内的Linux 2.6.12内核源代码注释文件，目前该内核源码注释已经托管到CSDN CODE。

CSDN CODE地址：https://code.csdn.net/chenyu105/linux_kernel_2-6-11-12_comment。

谢宝友（右二）和同事们的合影

下面是采访内容整理。

CSDN：先请您对自己和您的工作做下介绍。

谢宝友：我是一名老中专，1996年毕业于四川省税务学校税收专业，现供职于中兴通讯操作系统团队，对操作系统内核有较强的兴趣，专职于操作系统内核已经有六年时间。

在日常工作中，主要工作是对Linux内核进行分析，解决遇到的标准内核故障；并向项目组提出应用程序优化措施。当然，团队有时也会抓壮丁，去帮其他项目做一些虚拟化、Android相关的工作。我现在还没有达到“内核菜鸟”的水平，因此也就没有将关注的领域扩展到其他方面。在工作中，自己也从头编写过一款自主知识产权的嵌入式操作系统。

CSDN：您是从何时开始做开发相关工作？现在还写代码吗？

谢宝友：从1999年开始，我就拥有了第一份正式的“程序员”工作。十五年来，从来没有离开过代码，算是一个标准的程序员吧。有几次转管理岗位的机会，我婉言谢绝了。林语堂老先生在《吾国与吾民》这本书中提到：国人最大的特点在于“死”要面子。在技术领域，这实在没有必要。

CSDN：内核代码注释，您花了多长时间，又是如何完成的？其中有什么特别有趣或难忘的经历？

谢宝友：2008年，出于对操作 系统技术的爱好，我进入中兴通讯操作系统团队并开始接触Linux内核。在我的坚持下，部门领导把我分配到了内核组。从此，开始了长达6年的内核代码注释 过程。其中，前三年是收获最大的一段时期，这期间看了《深入理解LINUX内核》、《深入理解LINUX网络内幕》、《深入理解LINUX虚拟内存管 理》、《LINUX设备驱动程序》这几本经典内核书籍，共做了2200页、87万字的word笔记。

在学习内核的过程中，最难忘的经历是刚开始看《深入理解LINUX内核》这本书时，真有一种雾里看花的感觉。足足一个月，才将 《中断和同步》看完，这真是一种折磨。但是就在某一天，团队遇到一个非常诡异的故障，已经查了两个月，部门领导知道我当时正在看内核的书籍，于是报着试一 试的心态让我加入了攻关团队。没想到，故障真被解决了！其原因竟然还真的与中断处理流程有关（☺）。不由得让人感叹：天下没有让人白读的书。

CSDN：您最初为何要着手做Linux内核源码注释这项工作？是什么让您坚持了这么久来完成这项工作？

谢宝友：兴趣是最好的老师，乔帮主曾经说过：随心所想。如果我们真正喜欢某一件事情，就追随自己内心真实的想法，实践它。在智商、情商、逆商这三者中，最重要的是逆商。一旦决定做某件事情，就认真的做下去。

至于如何坚持下来，当然，最初做Linux内核注释工作不仅仅是出于对内核的爱好，也是工作的需要。后来发现，将注释的代码和 学习笔记共享给同事和朋友，是一件令人快乐的事情。用“赠人玫瑰，手有余香”来形容再合适不过了。同时，能够将个人兴趣与工作结合起来，对我们每个人，对 我们所在的企业来说，都是一件值得高兴的事情。

内核学习有一个非常陡峭的学习曲线。一旦越过了某个临界点，您就能发现不一样的风景，想不坚持下去都不容易。非常庆幸的是，在学习半年后，我找到了这样的临界点，在学习和工作中，都找到了一种快乐的感觉。

这项工作主要是利用业余时间完成的。几年来，每天晚上一般会抽出两到三个小时来看书、读代码。可惜的是，在工作三年后，随后工作任务越来越重，最近三年的时间不能保证每天抽三个小时来做这件事情了。

CSDN：您是独自完成的这项工作吗？有没有小伙伴陪您一起做？

谢宝友：目前已经公布的针对Linux内核2.6.11.12的代码注释，以及今年内会继续公布的针对2.6.24版本的代码注释，主要是个人完成的。公布出来的目的，就是想更多的朋友参与进来。

这两年还翻译了《深入理解并行编程》一书，这是与鲁阳、陈渝两位兄弟一起合作完成的。

CSDN：您在开源方面还有哪些参与？

谢宝友：我是中国开源软件推进联盟专家委员会的成员。中国开源软件推进联盟是在工信部指导下，由业界著名企业合作成立的非盈利性组织。个人接触开源软件就是从进入中兴通讯操作系统团队开始的。

CSDN：您如何评价中国目前的开源环境？您认为中国开源若要取得良好发展还需要在哪些方面做出努力？

谢宝友：在中国，无论是企业还是个人，生存压 力都不小，国内开源环境还算不上理想，这和国情分不开。中国开源若要取得良好发展，离不开企业和个人的努力。对个人来说，参与开源能够提升个人技能，扩大 个人在业界的影响。实际上，我所了解到的国内一些年薪百万左右的大牛，正是凭借自己在开源社区的贡献才被猎头关注到的。

因此，个人认为首先需要企业和个人得到发展，减轻生存压力；其次，需要企业和个人转变开源观念，这是中国开源取得良好发展的两个重要因素。

CSDN：程序员常自我调侃为“码农”，这个身份带给人很多的痛苦和欢乐，您如何看待这一职业身份？

谢宝友：这个问题应该怎么回答呢？在刚进入这一行时，那时“码农”还不是“码农”，叫“程序员”。在成都，每个月可以拿800至2000的高薪。在北京，则可以拿到4000左右，这足足能够在知春路买一个多平方的房子了。不知道从哪一天起，“程序员”变成“码农”了，甚至听说变成“码灰”了:)

但是请听我讲一个故事。2000年的时候，我的老板曾经说：“现在的软件行业，看起来还不错。但是要不了两年，就会成为夕阳产业，甚至雪崩，和房地产一个结局！”。

业界也比较流行30岁、35岁程序员的说法，但是我也听说：目前世界上还有这样一个程序员，他做了30年软件，至今还在做磁带 机！IBM的paul，就是那个维护Linux RCU（Read-Copy Update）的大牛，他也做了31年软件了。前两年他还在感叹很难找到一个完整的半天时间，舒舒服服的写点代码。

再举一个国内的例子吧。我最好的朋友，吴涛先生，那个发明易语言的家伙，他91年就在Z80上编写软件了，这个月我还和他讨论过代码方面的事情。

也许，我们可以把这些35岁的故事当成谣言！

作者：王殿进 苏慧

本文转载自：http://code.csdn.net/news/2819178

技术人攻略访谈二十五：运维人的野蛮生长

作者/技术人攻略
关注技术人的个人成长
讲述技术人自己的故事
传递技术梦想。

导语：本期采访对象邵海杨@海洋之心-悟空，UPYUN运 维总监。第一波互联网潮流兴起以来，Web运维作为一份职业已经存在了十几年。在普遍的印象里，运维人总是和“辛苦”这个词划上了等号。飞增的流量带来网 站规模和复杂性的提升，在高压环境下生存的运维人，谁不曾为服务上线彻夜坚守？谁未曾被半夜报警紧急叫醒？谁不是问题出现后第一个知道，问题解决后最后一 个离开？O'Reilly出版的《网站运维：保持数据实时的秘密》这本书的第一章，对运维职业做了这样的描述：Web运维没有定义好的职业路径，也没有教 育能够造就懂得运维Web基础结构的人才。追求Web运维这个职业，你将成为一名拓荒者。

想征服这块荆棘丛生的领地，成功的拓荒者不仅需要掌握坚实的基础，更少不了长期的坚持和探索的勇气。从毫无章法到游刃有余，邵海杨用了差不多10年 时间，当他可以把Linux玩得如同庖丁解牛（把Linux裁剪到12M），自然也能轻松玩转运维。于是我们看到了一个快乐的运维人，在UPYUN这家以 云服务为核心业务的公司，3人的运维团队用高度自动化的方式

管理着500台服务器，不仅能保证系统的可扩展性，还能实现平滑的升级和问题的快速定位。在荒 地上辟出新路，和邵海杨对Linux的热爱密不可分。见面那天，我们不幸选择了一家非常吵的餐馆。为保证录音效果，他扯着嗓子、眉飞色舞地吼了两个多小 时，说他怎么看几百台Linux在他的控制下跳舞，我眼前也瞬间浮现出几百只小企鹅跳舞的奇妙画面。他希望能传递运维的正能量，就和攻略君一起来看这段运 维人的拓荒历程吧！

- **技术人攻略：能否介绍一下你是如何把嵌入式Linux的思想应用到了运维领域？**

我98年进大学开始接触Linux，到04年加入台湾威盛之前，已经玩了6年Linux。包括各种服务器配置，IDC机房的网络，LAMP组合 都驾轻就熟，自认为玩得还不错。可接触了嵌入式Linux之后，才发现原来Linux还有很多奇妙的玩法和专用领域。之前只了解上层应用，对Linux本 身的印象还停留在几百兆的一张光盘和一堆服务配置，其实根本没有入道。

在威盛的工作是做车载嵌入式系统，属于定制化的Linux系统，去掉了不需要的硬件模块和驱动，封装出一个很小的、精干稳定的操作系统。做完这个系统之后，我对Linux的每一部分都了解得很清楚，像庖丁解牛一样，能把Linux做到12M（2005年用16MB的DOM盘）。

加入UPYUN以后，发现业务和硬件都是自由、自有且可控的，很适合为它打造一个专有系统。于是我把嵌入式的思想用到了运维中，打造了一个 86M的专有Linux系统。这个系统可以灌到4G或8G的U盘里，当上架新机器时，只需插上U盘，配合执行一些脚本，机器就摇身变成存储或CDN服务 器。UPYUN的500台机器都采用这种方式管理，现在哪怕一天部署300台也不成问题，极大提升了运维效率。

采用U盘系统的好处不仅在于快速安装，更重要是实现系统的平滑升级。有了U盘以后，可以把系统镜像写到U盘里，升级之后，再把镜像写回到磁盘， 通过两种媒介的切换，可以保证3分钟内整个系统焕然一新。采用光盘网络安装的方式虽然也能提升效率，但只有硬盘一个载体，无法及时给正在运行的操作系统升 级。我们的CDN分布在全国，数量多又分散，光盘网络安装的方式更适合集中式机房。

更酷的是，由于系统和安装的标准化程度非常高，系统又很基础，没有敏感数据，我们规范化和简化了上架流程。后续采购新机器，只需将定制好的U盘 快递给经销商，经销商用程序生成一个硬件检测报告，截图发给我们确认，证明这台机器OK，直接就可以发到机房上架。这省却了原来经销商先检查一遍硬件，打 包快递给我们，我们拆封检查、灌系统，再打包发给机房的中间环节。运维做到连机器都不碰，把人解放了出来。

- **技术人攻略：运维工作普遍很辛苦，你却能做得如此快乐，有哪些经验可以分享？**

流程比补位更重要，方法比拼命更重要。UPYUN运维团队只有3个人，但很早就做了流程规范和脚本处理，从 最初几台到现在的五百台机器，再到将来的五千台、五万台也都是用同样的方式管理。国内很多公司对运维的认知度不高，导致业务量上去之后，用堆机器的方式快 速抢占市场，运维也只能靠人力堆，24小时待命，事后救火，自然会觉得辛苦。

运维想做得轻松，首先要做到自动化，其次是监控常态化，然后是性能可视化。服务器不会无缘无故出问题，犯病之前肯定有征兆。用监控系统做连续的 健康检查，会很容易发现故障触发原

因。新出现的问题要及时增加监控数据，例如一台机器上发现CPU过热报警，处理后会监控所有的机器是否有CPU过热的情 况。

自动化做好之后，再也不怕频繁部署，而且排查问题变得非常轻松。批量上架10台机器，其中9台没问题，1台有问题，那肯定是硬件问题，因为都是 跑的同一个脚本，通过人海战术部署就无法这么快定位。还有个例子，有段时间我们发现某两个机房的表现不一样，因为程序是统一的，把正常运作的机房的程序拷 过来，在表现有问题的那个机房机器上重新配置生成一下，如果问题仍然存在，那么一定是机房的原因。自动化的工具和流程可以最大程度地把人和机器隔离开，减少犯错误的机会，快速定位问题。

运维自动化不只是为了帮运维工程师节省精力，更重要是实现整个系统的可扩展性，这也是BOSS最关心的。如 果某个节点随时可以摘掉，运维工程师就没必要24小时待命，要是不能摘，一旦出问题，哪怕是三更半夜也得爬起来。要做到良好的可扩展性，需要运维工程师从 架构上去设计它。eBay的工程师曾说过，做任何架构都要考虑一个问题：如果负载扩大10倍怎么办？架构的扩展性一定要从系统设计之初开始做。当然，不是 说一开始就要考虑架构扩到100倍怎么办，要用进化的思想去看架构，分阶段进行容量规划。运维工程师虽然不怎么写程序，但是他们接触了许多非常优秀的软 件，如Apache、Nginx、LVS等，好的运维工程师一定有好的分布式理念。

所有这一切想实现，前提是要做好时间管理。当你忙得像狗一样，没有留下时间思考，就没有机会去深入细节。有时间以后，就可以去做工作上的优化，包括工具的使用、流程的优化、执行结果的监控等，还可以考虑人员的互备和管理。这一切都是环环相扣的，只有把细节封装好了才能在时间、资源、和人员上做到衔接。

- **技术人攻略：运维做得这么好，一定和你过去的经历有很大关系吧?**

我运维能做得这么High的前提是对Linux真的很了解。大学时机缘巧合，同学的表哥从美国回来，告诉我们Linux很火。当时对未来没有明 确方向，没有别的选择，就对准Linux这个方向努力。四年里看了大量UNIX的书，做了大量实验，包括编译内核尝鲜。在大学里已经把系统管理员、网络管 理员的基础打扎实了。

2002年毕业后找的第一份工作是在广州一家数据中心做系统管理员，主要做基础运维，帮助客户调试机器和上架。那时候服务器很少，就算在数据中 心，一个月才会有一、二十台机器进来。03年回了杭州，做过计算机老师兼网管，但心里还是割舍不下对Linux的这一份热爱，于是加入浙大网络，接触到应 用运维，并认识了现在UPYUN的创始人。

04年我去了台湾威盛，开始做嵌入式Linux系统。在这家公司待了5年多，不仅让我重新认识了Linux，并且拓宽了职业生涯。做到第3年的 时候我感觉遇到了瓶颈，一方面不想放弃技术，另一方面又想了解客户和外面的世界。迷茫的我当时向公司总裁请教人生的规划，他曾是3COM前30位员工，在 他的建议下我转做售前技术支持。原以为工程师比技术支持牛，但后来才发现技术支持很锻炼人，不仅要懂技术，还要懂表达，考验现场发挥和感染力。通过写大量 的方案，还锻炼了写作能力。

做了两年技术支持后，被公司提升到技术管理岗。一开始不懂管理，工作分不下去，我自己也做得很累。于是又去找总裁聊天，才发现我保护下属的做法 其实是害了别人不能成长，而自己也成

了公司的天花板。好的管理者应该做托板，把下面的人托上来。明白了这个道理之后，我改变了策略，把手头的工作整理成 WiKi文档，交给下面的人去执行。多出来的时间用来跟踪新技术，做前瞻性的研究，并且把学到的东西分享给大家，帮助下面的人快速进步。

常有人问我，你把知识都分享出去了，会不会担心被取代。我个人认为这种担心大可不必，丰富的经验和阅历是偷不走的。纸上得来终觉浅，绝知此事要 躬行，我分享自己思考后的结果，别人想倒推我思考的过程仍然需要经验。就像大家每天都在吃饭，可为什么只有一小部分人成为美食家呢？因为这些人会用心去思 考，深入进去，方能总结一些门道出来。一个真正的强者，不是摆平了多少人，而是看他能够帮助多少人。我的从业经历也证明，如果把事情做得漂亮，不让自己成 为公司的瓶颈，反而会获得更多信任，就算离开依然互相尊重和感激，甚至在以后的人生路上，遇到困难别人也会帮助你。

- **技术人攻略：你过去一直在嵌入式领域发展，为什么会选择进入互联网行业?**

互联网蓬勃发展起来以后，围绕Linux出现了很多创新的技术，我骨子里对互联网很憧憬。嵌入式Linux做的往往是实体产品，对性能、并发性 没有考量，所以体会不到Linux在互联网如火如荼情况下的那种威力。我很向往操纵成百上千台机器的感觉。2010年加入了国内较大的一家在线客服系统， 对实时性，大并发要求很高。那时的工作是重新设计架构，提升性能，最后用原来一半的机器实现了同样的业务负载，但却离我最初想操控成百上千台机器的梦想有 点偏差。公司的业务规模无法让我完全施展自己的才华，于是重新做了选择。

那时在凤凰古镇待了两个礼拜，想自己未来的方向。98年选择Linux，是听了前辈的建议，这条路是选对了。工作多年以后，有了自己的阅历，就要结合自己的思考，去想未来在哪里。思考的出发点其一是兴趣爱好，其二是找到自己的自豪感来自哪里。我一直认为运维工程师就是让Linux跳舞的人，当我操纵几百台机器，整齐划一地做一件事情，那种感觉特别棒。

我想清楚了自己未来的方向是做云上的运维，选择加入UPYUN有几个原因，一是可以实现我操纵几百、几千台机器的梦想；二是我过去帮别人做架构 的项目，都转向了阿里云，我意识到云的发展一定会给传统运维带来打击；三是我比较擅长做业务的抽象和自动化，对DevOps有自己的想法；再加上 UPYUN几位创始人都是浙大网络的同事，在正式加入UPYUN之前，我就以顾问的角色帮他们解决运维和架构上的问题，彼此之间很信任。如果凭个人能力， 可能要等到四五十岁才能出来创业，但寻找到志同道合的人，大家能力互补，就可以在年轻的时候去追寻自己的梦想。

- **技术人攻略：云计算领域竞争激烈，你怎么看待UPYUN专注的云存储市场?**

UPYUN提供静态文件的存储和CDN加速服务。单从CDN来说，有老牌的网宿和蓝汛；单从后端存储来说，网盘大战早已白热化了。为什么 UPYUN能在夹缝中生存，因为我们走了一条服务路线。网宿、蓝讯只做CDN，不做后端存储；网盘只做存储，但目标是做用户的大数据分析，不会为网站做加 速。UPYUN将存储和加速结合起来，企业不需要利用多套架构，把东西放在网盘上，再找网宿和蓝讯去加速。单从CDN角度来说，网宿和蓝讯的带宽是混合模 式，包括了视频点播、流媒体等服务，导致高峰期带宽特别拥堵。而UPYUN是针对网站和移动应用的静态资源做

存储、处理和分发加速，目标人群高度统一，应 用类型有很多共通性，高峰期不会出现严重的带宽冲突。

UPYUN早期的创业方向是做图床网站又拍网，在图片领域有着7、8年技术积累。比起大而全，我们更擅长做小而美的事，同时也和专业的第三方公 司合作，例如DNSPod，Takling Data等。我们的理念是，行业之间要抱团，专业的事交给专业的人做，不然大家都在浪费时间和精力，还要提防被BAT干掉。

- **技术人攻略：你怎么看待DevOps，运维和开发的关系应该怎样平衡?**

DevOps国内大家提得多，用得少。难度在于思路上的转换，运维自动化是一个结果，要做到这一点，首先需要抽象业务模型。以业务软件性能监控 为例，如果软件工程师在程序中插入很多的钩子或探针，就可以统计出数据来，不需要运维费心监控；软件工程师在设计程序的时候，考虑到了分库分表，考虑到了 大并发和分布式的设计，运维就可以水平扩展机器。开源软件但凡名气大的，程序日志信息非常详尽，可以通过标准的syslog或者日志去监控。但根据我接触 的大多公司和工程师的情况，大家都忙于实现业务功能，连个文档甚至注释都不愿意写，更别提能够考虑这么周全了（UPYUN团队这方面做得不错），所以才需 要做运维的去补位、去优化流程。运维苦逼的公司，软件工程师也幸福不到哪里去，这种负能量是相互传递的。

至于是开发人员学会运维，还是运维人员学会开发，在我看来是殊途同归。高级软件工程师会测试自己的程序，知道性能指标是什么情况，但如果正好遇 上一个三流的运维工程师，程序性能上不去，那么这个软件工程师可能会自己去找原因，这样他就做了运维工程师的工作。另一种情况，一个三流软件工程师的程序 到了我手里，因为已经做了性能监控，第一时间就知道程序跑得好不好，如果程序出现死循环或者内存泄露，我会告诉这位工程师程序有bug。但如果软件工程师 很忙，那么我就要去把这个bug查出来。自己的修炼功底很重要，其次是要去寻找小伙伴，发现问题的时候如果能坐下来互相学习和探索，就能学到更多知识。

我们现在运维做得很好，可以倒逼开发，让他们加快新业务模型的开发，让公司加快业务增长速度，因为再上一百台、一千台机器都是分分钟的事情。运 维人员的强势，是要通情达理，站在全局的角度上，才能说程序员想听的话，说老板听得懂的话。我加入UPYUN之后，一方面做了很多内部分享，把我的理念毫 无保留地告诉大家，帮助大家提升工作的效率。另外用实际行动把运维自动化、标准化做出来，去年花了三个月做嵌入式Linux原型，一个半月测试稳定性，然 后用一台新机器放到集群里，性能监控发现新架构的机器响应、负载都有很大提升，然后就开始逐渐推广更新。

- **技术人攻略：运维领域有什么新趋势?**

虚拟化会越来越大行其道。现在运维能够让几百台机器跳舞，但还不能让几百个应用跳舞。举个例子，做日志分析的Hadoop集群晚上启动，而音频 处理或Python服务器白天会吃紧。要平衡物理机器上的资源利用率，只有依靠虚拟化，让同样的机器在白天变成Python服务器，晚上变成Hadoop 服务器。我们打算采用Docker做虚拟化，这个技术去年才诞生，但业界已有相当多公司对它抱有很大兴趣。

运维工程师有必要学一门能打通前后端的语言，如Node.js加Python。未来的应用会越来越轻，后端放在云里，前端是个浏览器。要从两个 角度去思考这个问题，首先JavaScript是唯一的浏览器原生语言，世界上几十亿的设备都在运行浏览器的时候，想想看它有多么重要。其次，前端用浏览 器这样轻巧的东西，后端必须有云的支持，这决定了运维人的职业生涯应该往云端靠。往云端靠有几个途径，作为开发者要了解BAT之类的开发平台，或者是加入 做云的公司，去做云上的运维和云上的开发。JavaScript会越来越大行其道，学一门能打通前后端的语言，能获得最高的学习效率。

- **技术人攻略：能不能给新人一些学习Linux的建议?**

任何技术都是竞技活，一定要多做、多练。我自己学Linux的时候是什么都玩，一开始的时候装系统、装Apache环境、装MySQL，也用 PHP写论坛，聊天室程序，踏踏实实做个完整的项目就是最好的锻炼。印象最深的就是2001年那会，编译内核加入了对reiserfs和ext3文件系统 的支持，那时的2.4内核可以裁剪到500KB，太有趣了。到了大四那年出来个Soft Raid，我们又开始编译，硬盘不够就从同学的机器上拆出来一个硬盘，用两块硬盘组成阵列做实验。

学习过程中不能老是跟着别人的脚步走，要有自己的思考，才能发明新东西。所谓的技术难题就是对未知的茫然，任何你不知道的东西都觉得是天大的东 西，捅破了就明白了，但关键是这个捅的过程，是自己捅、还是别人捅。如果遇到问题就搜索，对别人总结出来的方法不加思考地应用，很可能会影响自己的思考和 成长。我们那时没有这么发达的网络，也没有那么多的教材，很多东西都是要自己反复做实验，调参数，然后走在路上、吃饭时、睡觉时思考琢磨才能整明白。现在 条件这么好，我不反对凡事上网搜索，但一定要做好笔记，整理心得，要真正转化成自己理解的内在的东西。这跟古人的“学而不思则罔，思而不学则殆”是一个道 理。

多看书，看好书，如何看书也很重要，要学会查漏补缺，细品慢琢磨，我个人建议不光技术书，还有情商、管理、哲学的书都要看，做人一定要文艺（我 喜欢读《泰戈尔诗选》《明朝的那些事》）。少抱怨，要懂得感恩和回馈，这样人生才是精彩的。有机会尽多参加甚至组织开源活动，锻炼自己的口才和交际能力， “教”是最好的”学”。经营好自己的影响力，可以交到更多的朋友。推荐看《禅》道和《了凡四训》

《UNIX编程艺术》这本书要重点推荐，我做了很多年总结出来的东西，人家30年前就总结出来了。里面提到的“模块原则”、“分离原则”、“吝 啬原则”、“生成原则”、“扩展原则”，还有很多先进的理念都让人叹为观止。如果想要对Linux本质有深入理解，推荐去打造一个自己的Linux版本。有个开源项目叫LFS（厨房里的Linux），给你一本菜谱，告诉你打造Linux需要哪些软件，可以照着这本菜谱把Linux从头到尾编译一下。此外， 我们还要学会时间管理（不要告诉我你都会利用好时间），推荐《暗时间》和《把时间当作朋友》。

一定要找到自己学习的动力是什么，我招聘新人的时候对个人技能看得不重，更关心他的动力和热情来自于什么地方。在 威盛的时候曾招聘过一个高中生，他之前因为贪玩没考上大学，知道生活的艰辛后开始学Linux。公司原来从没招聘过高中生，他进来之后非常努力，成长非常 快。我在他身上看到一点自己的影子，高中一开始我成绩不错，以为凭着聪明就可以轻松学得不错，直到高三才明白，原来那些表面上的天才背后都下了功夫。那个 黑色的七月给了我很多教训，我再

也不认为自己是天才，而明白了要靠脚踏实地努力才能得到自己想要的东西。人在历过挫折以后，能够爆发出的能量是很惊人的。

人生就是一场修行，我们都是半杯水，这才有了人生存在的意义。不自卑，不骄傲，寻找互补，越努力，越幸运，做最好的自己！

作者：@devlevelup

本文转载自：http://blog.segmentfault.com/devlevelup/1190000000464916

年后跳槽那点事：乐视+金山+360面试之行

作者/吕大豹
前端懂交互，谁也挡不住
http://www.cnblogs.com/lvdabao/

有段时间没有更新博客了，年后经历了换工作以及生活中的其他各种事情，生活节奏被打 乱了一阵子。还好一切都平息下来了，我也能在这个周末的午后，静下来码码字了。在做去年的总结时就打算好年后来了换份工作，并为此做了一些准备。说说我的 准备吧，我的目标是互联网公司，同时作为一名毕业还不满一年的新手，我也知道自己的短处在哪里。因此在年底的时候就开始收集整理一些基础知识，前端面试题 什么的，网上一搜一大把，顺着能找出来好多，同时也在博客发表了一些【面试必备】文章，虽然没几篇。。。另外去年也一直在学习AngularJs，同时也 赶在面试前，在我的github上发布了第一个自己的项目，这些都是为了给自己增加点筹码，(*^__^*) 嘻嘻……

仓促的乐视之行

年后来了面的第一家是乐视，乐视刚刚上市，势头正猛，是个不错的选择。年后刚来上班几天，就在技术群(javascript后花园116366053，我最崇拜的文叔领导的) 里看到有人发的乐视的招聘信息，而且是乐视前端带队大哥亲自来发的。本来我计划再好好准备一阵再投简历的，因为自己的基础知识还没有系统的整理一遍，但瞅 着机会就在眼前不容错过，于是冒冒失失的投了一份简历过去。真的是冒失加仓促，我的简历上都忘了写掌握的技能，对方让我补充上再发一份，我不好意思的重新 整理了一下简历，真囧。幸好面试官比较开明，看的出来是真心想招人。

不久就接到hr的面试通知，于是我胸中抱着小鹿乱撞来到了乐视，感觉环境还不错，就是挺拥挤的，一个大厅里一排排坐满了开发人员与行政人员的混 合大队，果然是一家高速发展的公司啊，人都快放不下了。随后hr给了我一份笔试题，我就开始做了。不得不说题可真多，足足做了一个小时，做完脑袋都晕了。 其实都是些基础题，考察js的作用域、原型、类型转化什么的，我不太会答的是前端页面对于大数据的优化处理，有些直接空了没填，我觉得笔试题也就是个基本 测试吧，关键还得看面试。

笔试完后一会，就开始了技术面。这个时候我刚做完题还有点晕，加之自己也紧张，可能面部表情比较不自然吧，被面试官看出来了，问我紧张吗，我说 有点。他递给我一瓶可乐说先喝着，然后看我答的题，同时跟我聊了些题外话，我渐渐放松了下来。不得不说这位面试官还是挺好的，懂得让选手先放松，我心里也 挺感激。然而接下来将近一个小时的面试着实

是把我虐惨了，我深深感觉到了自己是有多没准备好，比较浅一点的还能答上来，再往深像DOM接口，事件，跨域 等，基本上一问三不知了，面试官估计对我的分量比较怀疑了，当他问我js都有哪些基本类型时，我顿时感觉慌张了。这好比是体检视力时，人家指着最大的那一 个问你朝哪个方向。基本已经在判断你是不是什么都不会。当面试官给我来了句“你毕业这么长时间了，这些都没搞明白，你平时都在干嘛呢”，我顿时觉得没戏 了。后来我就说我平时看看书，学angular框架，刚刚开始玩github，他对github比较感兴趣，随后我展示了下我的项目，他看上去还挺满意。

其实平时忙着做项目，这些基本知识真的都被忽略了，没有系统整理一下真的一时想不起来。我心里这么想但也没敢说，都发挥成这样了，真没脸面说什 么了。最后面试官给了我这样的结果：“今天这么聊也对你还是没能整体了解，这样吧，晚上我给你发个代码，你来看一下，给我讲讲代码是干嘛的”。

真在我意料之外，我寻思面试官是这么想的：这小子基础知识答的这么屎，博客、github倒还写的像模像样，到底是什么货色真不好把握，再给最后一个机会吧。

所幸的是我不辱使命把握好了这次机会，他给我发来两个文件，一个是jquery插件，一个是seajs模块，小弟虽然理论知识不太行，平时代码 写的还是不少，看了下基本难不倒我，于是无比仔细的在源码上逐行注释，每段代码的功能都详细描述。同时还指出了代码中的一些缺陷，以及改进的办法。我自认 为做的还不错。good job~

果然我对代码的理解让他满意，通知我初试通过了。随后我又去了一次进行复试，面试的应该是部门负责人，没问什么技术题，基本了解了下情况。只问 了一个关键性问题：加班能接受不，我心想这不是小菜一碟么，说能。他又问，一周加班五六天能吗？我菊花紧了一下，这不就是说天天加班吗！但既然问了，我显 然不能示弱，投去果敢的眼神，说，能，我身体倍儿棒。他点点头。

乐视的面试就这样结束了，年后第一面，初试虐的够惨。我也深知自己基础方面的缺陷，赶快买了高程第三版来看。另外再透露下，初试的好些题都是高程上的，估计面试官对高程很熟。

马拉松般的金山之行

面试的第二家公司是金山网络，靠猎豹浏览器容光焕发的公司，去年年会发金条的就是他们。也是走互联网路线，当然也是个不错的选择。在拉勾网投的 简历，很快就通知面试了。这次的面试也是挺不容易的，为什么说马拉松般的呢，我下午四点过去面试，从笔试到hr最终面一气呵成，持续四个小时，八点钟离开 公司。下面来细说说。

进去公司后前台MM给了我份笔试题，我就开始做了。内容无外乎也就那些，给段代码写写输出什么的。不同的是金山对HTML5和CSS3有较多考 察。大概四十分钟后做完了，等待面试。有两位面试官同时面我。其中一位向我介绍另一位：这是网络上大名鼎鼎的XXX大神。本期待我会投以闪闪的目光并说 “哦！原来XXX大神就是你啊！”。尴尬的是小弟孤陋寡闻，根本没听说过这位大神，说了句不知道，尴尬的笑。他一脸黑线。

被两个人面的感觉也难过的，一会问这个一会问那个，两个人跳来跳去，有时我这个问题还没考虑完，就又问另一个了。他们自己貌似也没什么逻辑，从 东扯到西，想起什么问什么。原型、作

用域这些基本是必问，但不单单是问概念，还问些相关的东西。比如原型上定义的方法和静态方法有什么区别我就没答上来， 丢人了。另外就是HTML5和CSS3，就我用过的讨论了下，什么sessionStorage、localStorage，FILE，xhr2。感觉金 山网络这头用到的新技术还挺多，往深了问我就答不上来了。

整体感觉初试比在乐视顺利些，面试完后两位示意我稍等下一轮面试。等待期间，我坐在椅子上，观察了下工作环境。员工挺自由的，有零食、水果提 供，不时有人来拿。每个桌上都放了一碗糖。我有点饿了，吃了一颗。。。还看到了不少外国友人，好像是员工，竟然还有包着头巾的阿拉伯国家的，真是让我大开 眼界。

二面的面试官在开会，我等了有半个多小时，终于来了。而且竟然是外国友人（应该是，反正不是汉族的面孔），操着不太流利的普通话。聊了些在现在 公司的基本情况，个人生活习惯啥的，还有在大学期间的情况，没详细问技术，回答这些，我向来比较拿手，操着流利的普通话回答的头头是道，他频频点头。他好 像有些疲惫，问了一会就走了，让我等候下一轮。我有点纳闷，不知道这一轮到底算什么面。

这一稍等又将近半个小时，接下来是部门老大面，我有点压力。不过面试官比我想象的随和好多，问的问题很广泛，问大学学过哪些课程，我是软件工程 专业的，什么操作系统、数据库、算法通通都说了一遍，问我成绩如何，四级多少分，我说都还不错。看了这位老大比较注重基础。然后他拿出手机，给我展示了下 flapybird，问如何实现，我大体说了说，然后他问如何优化性能，我也没做过canvas程序，随便说了说渲染性能。。。然后还问了些细枝末节的， 比如打算几年后结婚什么的。。。

这轮面试通过了，老大让我稍等，下面hr面试。这一稍等，哎。。。这个时候已经是七点多了，我的肚子都咕咕叫了，看看公司四周员工都还在忙，没有一个下班的。顿时感觉到了这是一家多么有“活力”的公司。

跟hr随便聊了聊，知道我要去的部门是猎豹浏览器，明星产品，环境挺不多。然后就回家等通知了。她也知道我等这么久饿了，给我抓了一把零食，体现出了细致入微的人文关怀，我直接装口袋里了。出公司的时候一看表八点了，不过还好整个流程一趟搞定，效率也挺高。

顺风顺水的360之行?

经过两次面试后，我算是找到了一点感觉，而且这期间也一直的马不停蹄的学习，基础知识整理的差不多了。恰巧有一个同学在360，就让他帮我内推 了。跟之前两家相比，360的名气以及近两年的势头显然更凶。而且我大三的时候有面试360实习被刷的经历，现在回忆起来还隐隐作痛。所以此行也是不能掉 以轻心的。

效率很高，几天后我已经坐到360楼下的咖啡吧等待召见了。上来依旧还是笔试题，笔试的知识核心都还是那些（作用域、原型、闭包、css布局原 理、性能优化），只不过360的题明显难些，需要真正理解原理才能解出来。题量相对较少，我半个小时就做完了。有几道题不太确定，我蒙了个答案，后来发现 竟然蒙对了，天助我也。

然后来了一位长相和善的大哥来面试我。聊了聊基本知识，拿着我做过的项目问了问如何做的。聊到基础的时候我感觉明显比头两次顺利多了。也不知是 运气好，问的都是我会的，还是我这几天的努力有成效，总之是万幸啊~我在网上也看了些面试技巧啥的，要尽量把话题引到你擅长的地方，于是我就多谈了下我 github上的那个东西是怎么做的，他表示满意，我不禁暗喜。有一个地方卡了一会，就是在手机上加载图片列表时的性能问题，他问我有什么办法。我说了延 迟加载，说了实现方式。但好像不是他想要的答案。我突然想到以前听分享会有人提到动态删除节点，在浏览器内只保存两到三屏的节点，这样可以大大减轻浏览器 压力。我这么说了，他点头。好，猜中了他想要的答案，然后又问我如何判断哪些节点在屏幕内，哪些在屏外该删除了。我说了几种标记的方法都不是他想要的。最 终，在他的各种提示之下，我还是没答上来。。。他说你不是学过算法吗，可以用二分法查找。我一惊压根没想到他要算法，然后边听他解释边点头，好像我听懂了 似的。

整体下来应该是顺利的，我自己都感觉还行。果然，过了一会，hr来找我了。是的，接下来是hr面。我有点奇怪怎么就一轮技术面呢，当年面实习生 还两轮呢。。。难道我太优秀了直接拍板了？不可能吧，我又不是天才樱木。不管那么多，hr面最好应付了，把自己的真实情况老老实实交代就行了，反正也没什 么污点。完事后，就回去等消息了。

过了几天，hr的电话来了，说通过了，聊了工资，明天发offer。欧耶！没想到进360竟然这么顺利！顺的我都不敢相信了！

果然不该相信啊！故事还没有结束，第二天了，hr来电话了，说offer还不能发，说部门比较乱，那天部门负责人没有见我，需要再来一轮面试。 我这个忐忑啊，各种猜测，难道后悔录用我了？工资给说高了要降一降？怀着乱七八糟的心情，我又来到了楼下咖啡吧等候召见。哪知这天特别让人火大，面试我的 人迟迟不来，我给hr打电话也没人接，足足等了一个小时，给hr打通了电话，我面带微笑无比温柔的说，您好，我还没有接到面试。纯爷们儿，这点肚量小意 思。一会面试官来了，我迅速调整好状态。

说是部门负责人面试，但我看面试官一点也不像个领导。穿着普通，发型普通，年龄也不算大，说话也挺随和，一点没有派头。我们聊的内容，也是基础 编程知识，从js到css hack，我感觉面试官还是一线程序员，不像hr所说的领导，看来这个部门确实有点乱%>_<%。今天运气也比较好，问的都是我会的。而且我 又使了那一招，把话题拉倒我擅长的部分，面试官比较实诚，被我给说的乐开了花。

一次挺愉快的面试。结束后其实我自己也有底了。好了，回去等通知吧。

我的感受

先说说结果吧，三战全胜，我陆续收到了offer。给开的工资也比较有意思，每个都比上一个offer多一点，看来他们也在竞争，于是我做了个艰难的决定，把前两个给拒了。

重点说说我跳槽的感受。首先，我对自己的技术是有谱的，不是什么大牛，也就是刚刚上了道的前端而已，事实确实如此。这也就反衬出来今年的招聘形 势确实比较好，各大公司都有新的布局，人才需求量大，连我这么个小兵都这么顺利，所以大家尽可以多多关注，微博上一些大牛天天在招人，我已经找到了自己合 适的归属，就不再贪多了。其次，这些大公司的面试真的没有想

象中的那么难，只要认真准备发挥正常，问题是不大的。但前提是，你自己要有谱，你在平时要有积 累，冲刺并不能解决所有问题。另外，心态也挺重要的，我经历的这三次面试，要么是准备的仓促，要么是漫长的等待，要么是意料之外的折腾，不论遇到什么情 况，都得应付的过来。

自己也不是什么成功人士，不谈经验了。最后，前端这个职位还是很有前途的，未来几年会更加火，把握吧，同仁们。

换工作的经历也让自己浮躁了一段时间，接下来会好好在沉淀下来，继续更新博客。照着去年做的计划，一点点积累。